МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧОРНОМОРСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІМЕНІ ПЕТРА МОГИЛИ

Факультет комп’ютерних наук

Кафедра інженерії програмного забезпечення

**КОНСПЕКТ ЛЕКЦІЙ**

**з дисципліни «Операційні системи»**

для підготовки студентів

Напрям підготовки: 121 – Інженерія програмного забезпечення

Галузь знань: 12 «Інформаційні технології»

Освітньо-кваліфікаційний рівень: бакалавр

Укладач: к.т.н., ст. викл. Г.В. Горбань

Розглянуто на засіданні кафедри *інженерії програмного забезпечення*

Протокол №\_\_\_ від «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 201\_ р.

Миколаїв – 201\_

**Лекція №1**

**«Поняття операційної системи. Історія ОС Linux. Командна оболонка в ОС Linux. Файлова система Linux. Основні операції для роботи з файлами та каталогами»**

**Поняття операційної системи**

Операційна система - це комплекс програм, який забезпечує управління апаратними засобами комп'ютера, організовує роботу з файлами (в тому числі запуск і управління виконанням програм), а також реалізує взаємодію з користувачем, тобто інтерпретацію команд, що вводяться користувачем, і виведення результатів обробки цих команд .

Без операційної системи комп'ютер взагалі не може функціонувати як такий. В такому випадку він являє собою не більш ніж сукупність непрацюючих електронних пристроїв, незрозуміло навіщо зібраних воєдино.

На сьогоднішній день найбільш відомими операційними системами для комп'ютерів є сімейства операційних систем Microsoft Windows і UNIX. Перші ведуть свій родовід від операційної системи MS-DOS, якій оснащувалися перші персональні комп'ютери фірми IBM. Операційна система UNIX була розроблена групою співробітників Bell Labs під керівництвом Денніса Річі, Кена Томпсона і Брайана Керніган (Dennis Ritchie, Ken Thompson, Brian Kernighan) в 1969 році. Але в наші дні, коли говорять про операційну систему UNIX, найчастіше мають на увазі не конкретну ОС, а скоріше ціле сімейство UNIX-подібних операційних систем. Саме ж слово UNIX (великими літерами) стало зареєстрованим товарним знаком AT & T.

**Історія ОС UNIX**

Потрібно згадати історію операційної системи, на засадах якої була створена Linux, - UNIX. Роботу над нею почав в 1969 році працівник компанії AT & T Bell Labs Кен Томпсон, який мав намір написати усічений варіант операційної системи MULTICS, робота над якою деякий час назад велася в цій компанії. В результаті вийшла операційна система, написана повністю на асемблері і отримала жартівливу назву UNICS (UNiplexed Information and Computing Service - примітивна інформаційна та обчислювальна служба). Потім для зручності система була перейменована в UNIX, так як кінець слова читається однаково в обох випадках. Згодом до проекту приєднався Денніс Рітчі, а потім і весь його відділ. У процесі свого розвитку UNIX був переписаний для новішого типу комп'ютерів і перенесений з мови асемблер на мову високого рівня C, який був створений Деннісом Рітчі для спрощення перенесення UNIX на інший тип комп'ютера.

Потрібно згадати історію операційної системи, на засадах якої була створена Linux, - UNIX. Роботу над нею почав в 1969 році працівник компанії AT & T Bell Labs Кен Томпсон, що мав намір написати усічений варіант операційної системи MULTICS, робота над якою деякий час назад велася в цій компанії. В результаті вийшла операційна система, написана повністю на асемблері і отримала жартівливу назву UNICS (UNiplexed Information and Computing Service - примітивна інформаційна та обчислювальна служба). Потім для зручності система була перейменована в UNIX, так як кінець слова читається однаково в обох випадках. Згодом до проекту приєднався Денніс Рітчі, а потім і весь його відділ. У процесі свого розвитку UNIX був переписаний для новішого типу комп'ютерів і перенесений з мови асемблер на мову високого рівня C, який був створений Деннісом Рітчі для спрощення перенесення UNIX на інший тип комп'ютера.

Операційні системи того часу не були зручними, тому UNIX швидко прийняли всюди - від університетів до великих компаній. Разом з операційною системою поширювався вихідний код, і користувачі могли змінювати або додавати нові можливості, тобто зробити UNIX максимально придатною для себе. На фоні такого буму створювалося безліч підверсій UNIX, які відрізнялися від оригіналу. Однією з найбільш вдалих стала розробка університету Берклі - Berkeley UNIX, перша версія якої називалася 1BSD (First Berkeley Software Distribution - перший розповсюдженні програмного забезпечення Берклі). Програмісти істотно доповнили систему новими можливостями і програмами, зробивши великий внесок в розвиток цієї операційної системи. Після цього багато розробників UNIX стали засновувати свої версії не на продукті компанії AT & T, а на версії Berkeley UNIX, і Berkeley UNIX став конкурентом оригіналу.

Протягом довгого часу UNIX розроблявся кожним програмістом для своїх потреб, тому програми, написані для однієї версії UNIX, могли не працювати на інший, і про комерційний успіх цієї системи говорити не доводилося. З цієї причини був виданий документ, який містив основні стандарти, яким повинні були слідувати розробники. Це підвищило сумісність версій UNIX, але не зробило впливу на версії BSD. Найбільш відомими версіями UNIX стали BSD, MINIX (розробляється відомим голландським професором Ендрю Таненбаум), SCO UNIX, System V (оригінальна версія компанії AT & T), Solaris (розробка корпорації Sun), XENIX (колись продавалася версія UNIX корпорації Microsoft) і, звичайно ж, Linux.

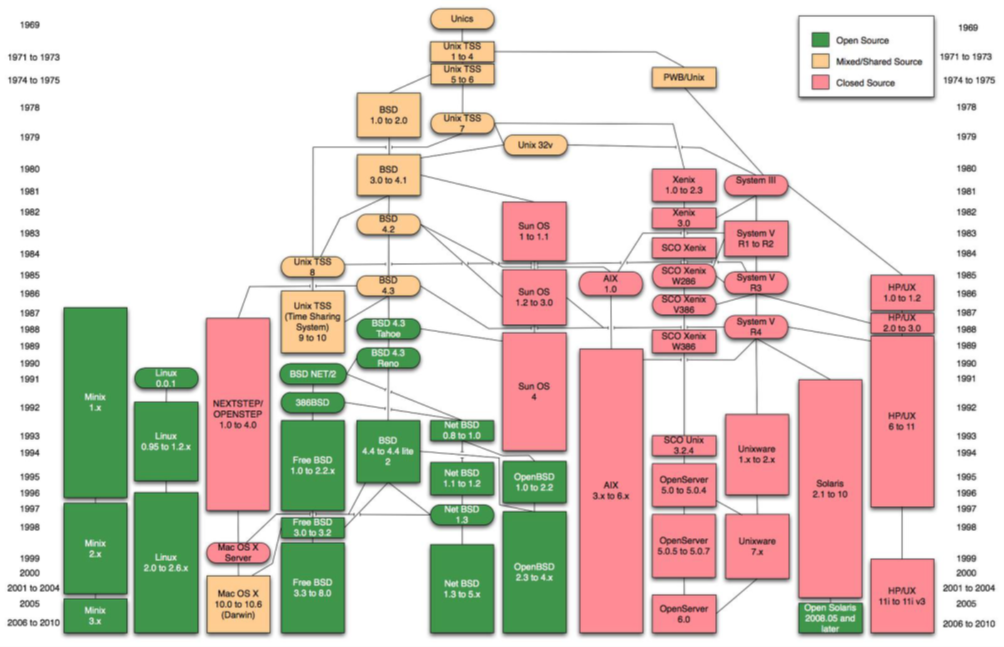


Рис. 1.1. Історія розвитку ОС UNIX

**Виникнення ОС Linux**

Історія створення Linux проста. Вона почалася в 1991 році. Фінський аспірант Лінус Торвальдс, придбавши пакет операційної системи MINIX, розчарувався програмою емуляції терміналу, що поставляється з нею. Він вирішив переписати її, не прив'язуючи до жодної операційній системі. Після деяких модифікацій почало з'являтися подоба ОС, а потім, коли вихідний код був викладений на сервері, проект став розвиватися завдяки зусиллям і, головне, ентузіазму програмістів, після чого він став повноцінною операційною системою. Проект взяв багато від існуючої вже тоді операційної системи UNIX. Навіть назва проекту було «зібрано» з імені розробника і букви X в кінці слова як свідчення схожості з UNIX. Назву Linux проект отримав не відразу. Спочатку він називався Freax як гібрид англійських слів free (безкоштовний, вільне володіння) і freak (чудний) із закінченням X, однак, коли Лінус Торвальдс викладав код на сервері, проект отримав своє остаточне ім'я і логотип (рис. 1.2).

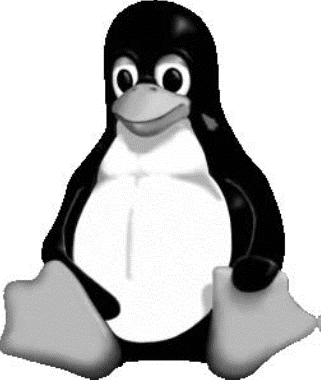


Рис. 1.2 - Пінгвін Tux - символ Linux

Після багатьох років після створення UNIX внаслідок тривалого розвитку і вдосконалення як самим Лінусом Торвальдсом, так і численними розробниками-добровольцями Linux стала повноцінним некомерційним клоном UNIX. Попутно з розвитком самої системи вона була перенесена на безліч інших платформ завдяки тому, що основна маса коду була написана на апаратно-незалежній мові C (в цьому випадку переписується тільки та частина коду програми, яка відповідальна за «спілкування» програмного забезпечення з апаратним забезпеченням комп'ютера). Вихід першої офіційної версії Linux відбувся в 1994 році. Вона містила всі необхідні функції, включаючи роботу з мережами. Ця версія була сумісна з UNIX, для неї листувалися багато програм. До розробки підключилося ще більше програмістів. Linux впевнено завойовувала авторитет. У 1995 році був зареєстрований товарний знак Linux, а в 1996 році вийшла версія Linux 2.0. До сих пір контроль над розвитком ОС зберігає за собою Лінус Торвальдс. Окремо слід відзначити принципи поширення Linux. Спочатку і до сих пір Linux поширюється як вільне програмне забезпечення за ліцензією GPL (General Public License - стандартна громадська ліцензія). Кожна людина може отримати доступ до вихідного коду операційної системи, доповнити, змінити що-небудь або використовувати код в своїх проектах.  Все це можливо за умови, що модифікований вихідний код буде так само доступний будь-якій людині, як і оригінал, і поширюватися по ліцензії GPL.

Зараз Linux працює на різних типах комп'ютерів і пристроїв - кишенькових комп'ютерах, мобільних телефонах, ігрових приставках, персональних комп'ютерах, серверах, суперкомп'ютерах та навіть музичних інструментах - завдяки своїй гнучкості і можливості перенесення на інші платформи.

**Основні характеристики ОС Linux**

В силу того, що вихідні коди Linux поширюються вільно і є загальнодоступними, до розвитку системи з самого початку підключилося велике число незалежних розробників. Завдяки цьому на сьогоднішній момент Linux - найсучасніша, стійка система, що швидко розвивається та майже миттєво вбирає в себе самі останні технологічні нововведення. Вона має всі можливості, які притаманні сучасним повнофункціональним операційним системам типу Linux. Наведемо короткий список цих можливостей.

1. **Реальна багатозадачність**

Всі процеси незалежні; жоден з них не повинен заважати виконанню інших завдань. Для цього ядро здійснює режим поділу часу центрального процесора, по черзі виділяючи кожному процесу інтервали часу для виконання. Це істотно відрізняється від режиму "витіснення багатозадачності", реалізованому в Windows 95, коли процес повинен сам "поступитися" процесором іншим процесам (і може сильно затримати їх виконання).

1. **Розрахований на багато користувачів доступ**

Linux - не тільки багатозадачна ОС, вона підтримує можливість одночасної роботи багатьох користувачів. При цьому Linux може надавати всі системні ресурси користувачам, що працюють з хостом через різні віддалені термінали.

1. **Свопування оперативної пам'яті на диск**

Свопування оперативної пам'яті на диск дозволяє працювати при обмеженому обсязі фізичної оперативної пам'яті; для цього вміст деяких частин (сторінок) оперативної пам'яті записується в виділену область на жорсткому диску, яка трактується як додаткова оперативна пам'ять. Це трохи знижує швидкість роботи, але дозволяє організувати роботу програм, що вимагають більшого обсягу ОЗУ, ніж фактично мається на комп'ютері.

1. **Сторінкова організація пам'яті**

Системна пам'ять Linux організована у вигляді сторінок об'ємом 4К. Якщо оперативна пам'ять повністю вичерпана, ОС буде шукати давно не використані сторінки пам'яті для їх переміщення з пам'яті на жорсткий диск. Якщо будь-які з цих сторінок стають потрібні, Linux відновлює їх з диска. Деякі старі Unix-системи і деякі сучасні платформи (включаючи Microsoft Windows) переносять на диск весь вміст ОП, що відноситься до непрацюючого в даний момент з додатком, (тобто всі сторінки пам'яті, пов'язані з додатком, зберігаються на диску при нестачі пам'яті) що менш ефективно.

1. **Завантаження виконуваних модулів "на вимогу"**

Ядро Linux підтримує виділення сторінок пам'яті на вимогу, при якому тільки необхідна частина коду програми, що виконується, знаходиться в оперативній пам'яті, а невикористовувані в даний момент частини залишаються на диску.

1. **Спільне використання виконуваних програм**

Якщо необхідно запустити одночасно кілька копій якогось додатку (або один користувач запускає кілька ідентичних завдань, або різні користувачі запускають одну і ту ж задачу), то в пам'ять завантажується тільки одна копія виконуваного коду цього додатка, яка використовується всіма одночасно що виконуються ідентичними завданнями.

1. **Загальні бібліотеки**

Бібліотеки - набори процедур, що використовуються програмами для обробки даних. Існує кілька стандартних бібліотек, використовуваних одночасно більш ніж одним процесом. У старих системах такі бібліотеки включалися в кожен виконуваний файл, одночасне виконання яких призводило до непродуктивного використання пам'яті. У нових системах (зокрема, в Linux), забезпечується робота з бібліотеками, які динамічно і статично розділяються, що дозволяє скоротити розмір окремих додатків.

1. **Динамічне кешування диска**

Кешування диска - це використання частини оперативної пам'яті для зберігання часто використовуваних даних з диска, що істотно прискорює доступ до часто використовуваних програм і завданням. Linuх використовує більш динамічну систему кешування: пам'ять, зарезервована під кеш, збільшується, коли пам'ять не використовується, і зменшується, якщо системі або процесу користувача потрібно більше пам'яті.

**Командна оболонка та інтерпретація командного рядка**

**Командна оболонка (shell)** – програма, що взаємодіє з користувачем за допомогою текстового інтерфейсу. Він називається інтерфейсом командного рядка (CLI). Оболонка дозволяє користувачеві запускати програми і виконувати команди операційної системи.

Оболонка інтерпретує введені користувачем команди, перетворюючи їх в інструкції операційної системи.

Всі оболонки мають схожі функції і властивості, в відповідністю з їх основним призначенням - виконувати команди користувача і відображати результати їх виконання.

Прикладом оболонки може бути, інтерпретатор команд **command** операційної системи MS DOS, або оболонка **bash** операційних систем UNIX/Linux.

Після старту оболонки, на екран виводиться запрошення до введення команд (зазвичай у вигляді знака долара **$**, якщо робота виконується в контексті обліковим записом користувача, або фунта **#**, якщо оболонка використовується під обліковим записом привілейованого користувача **root**)

Для операційних систем сімейства UNIX/Linux можливе використання декількох різних оболонок, що відрізняються властивостями і методами взаємодії з системою.

Найбільш поширеними є:

* **sh**- оболонка Bourne, класична оболонка для ОС Unix;
* **bash**- оболонка Bourne Again (GNU Bourne-Again SHell), найбільш поширена на даний момент , оболонка в середовищі ОС сімейства Linux;
* **ksh** –оболонка Korn, розроблена як розвиток оболонки Bourne з історією командної рядки і можливістю редагування команд;
* **csh**- оболонка C, яка використовує синтаксис популярної мови програмування C;
* **tcsh**- версія оболонки C з інтерактивним редагуванням командного рядка .

В системі може бути встановлено кілька різних оболонок, і для кожного користувача можливе використання своєї, оболонки, яка запускається за замовчуванням.

**Команда** - користувацьке введення у відповідь на запрошення оболонки. Команда Linux - рядок символів з імені команди і аргументів, розділених проміжками.

Команди часто супроводжуються одним або декількома параметрами, що змінюють їх поведінку, і додатковими, одним або декількома, аргументами, на які впливає команда. Тому більшість команд виглядають приблизно так:

**команда -параметри аргументи**

Аргументи надають команді додаткові параметри, що визначають її поведінку. В якості аргументів використовуються опціі і іменафайлов та каталогів.

Наприклад, командний рядок **ls-l file01 file02**містить команду ls, опцію-l і два імені файлів file01 file02. При використанні декількох опцій, їх можна об'єднувати.

Наприклад, варіанти наступних команд ідентичні: **ls -l -d**і **ls -ld**. Команди, які є частиною оболонки, називаються **вбудованими**. Наприклад, **cd**, **if**, **case**і т. д., Вбудовані команди можуть відрізнятися для різних варіантів оболонок.

Більшість команд використовують параметри, що складаються з одного символу, якому передує дефіс, наприклад: **-l**. Але багато команд, в тому числі команди з проекту GNU, підтримують параметри з довгими іменами, що складаються з слова, якому передують два дефіса. Крім того, багато команд дозволяють об'єднувати разом параметри з короткими іменами. У наступному прикладі команді **ls**передаються два параметри: параметр **l**, що вимагає використовувати довгий (long) формат виведення, і параметр **t**, що вимагає сортувати результати за часом (time) зміни:

**$ ls -lt**

Додамо параметр з довгим ім'ям --reverse, щоб змінити порядок сортування на зворотний:

**$ ls -lt --reverse**

Крім вбудованих команд, можливе використання програмних модулів, що представляють собою окремі виконувані файли, або файли скриптів, що містять послідовно виконуються рядки з командами оболонки.

**Файлова система Linux**

Ті, хто тільки освоюють систему Linux, можуть відчути розгубленість, дізнавшись про те, як в ній позначаються файли і каталоги, особливо якщо перед цим доводилося знайомитися лише з організацією роботи з файлами в операційній системі Microsoft Windows. Перш ніж приступати до вивчення системи Linux, необхідно зрозуміти, як вона влаштована.

Перша відмінність, яке можна помітити, полягає в тому, що в іменах шляхів Linux не використуються імена дисків. У світі Windows імена шляхів до файлів завжди містять літерні позначення дисководів, встановлених на персональному комп'ютері. В системі Windows кожен фізичний дисковий накопичувач позначається буквою, а кожен диск має власну структуру каталогів, яка використовується для отримання доступу до зберігаються на ньому файлів. Наприклад, в Windows часто можна бачити приблизно таке позначення шляху до файлів:

**C:\Users\Rich\Documents\test.doc**

Це позначення вказує, що файл test.doc знаходиться в каталозі Documents, який безпосередньо перебуває в каталозі Rich. Каталог Rich розташований під каталогом Users, який знаходиться в розділі жорсткого диска з присвоєної йому буквою C (зазвичай таким є перший жорсткий диск на персональному комп'ютері).

Шлях до файлу Windows дозволяє точно дізнатися, в якому фізичному розділі диска міститься файл test.doc. Наприклад, якщо файл повинен бути збережений на флеш-диску, то він може бути позначений буквою диска J. Після клацання на значку диска J, що знаходиться на робочому столі, автоматично відкривається доступ до файлу J: \ test.doc. Цей шлях вказує, що даний файл знаходиться в корені диска, якому присвоєно буквене позначення J.

В системі Linux використовується інший метод позначення файлів. Всі файли Linux зберігаються в єдиній структурі каталогів, званої віртуальним каталогом. Віртуальний каталог визначає шляхи до файлів, що знаходяться на всіх пристроях, що запам'ятовують, які встановлені на персональному комп'ютері, об'єднуючи їх в єдину структуру каталогів.

Структура віртуального каталогу Linux включає єдиний основний каталог, званий коренем. Каталоги і файли, що знаходяться під кореневим каталогом, перераховуються з урахуванням шляху до каталогу, використовуваного для їх досягнення, аналогічно тому, як це відбувається в системі Windows.

При цьому, наприклад, застосовуваний в системі Linux шлях до файлу **/home /rich/Documents /test.doc** вказує лише те, що файл test.doc знаходиться в каталозі Documents, розташованому в каталозі rich, який міститься в каталозі home. Це позначення не надає ніякої інформації про те, на якому фізичному диску персонального комп'ютера зберігається цей файл.

Складність, пов'язана з використанням віртуального каталогу Linux, полягає в тому, що в нього входять структури каталогів усіх запам'ятовуючих пристроїв на комп'ютері. Перший жорсткий диск, встановлений в персональному комп'ютері Linux, іменується кореневих диском. Кореневої диск містить ядро ​​віртуального каталогу. Від нього виходять всі інші частини віртуального каталогу.

На кореневому диску система Linux створює спеціальні каталоги, звані точками монтування. Точки монтування - це каталоги в віртуальному каталозі, за якими закріплюються додаткові пристрої, що запам'ятовують.

Монтування запам'ятовуючих пристроїв у віртуальному каталозі призводить до появи файлів і каталогів цих запам'ятовуючих пристроїв під точками монтування в каталогах монтування, без урахування того, що вони фактично можуть перебувати на інших дисках.

На кореневому диску часто фізично розташовуються системні файли, в той час як для користувача файли зберігаються на іншому диску (рис. 1.3).

Як показує рис. 1.3, на персональному комп'ютері є два жорстких диска. Один жорсткий диск пов'язаний з коренем віртуального каталогу (що позначено окремої косою рисою). Інші жорсткі диски можуть бути змонтовані в певних місцях віртуальної структури каталогів. У даній прикладі другий жорсткий диск змонтований в місці розташування **/home**, яке представляє собою те місце, де знаходяться призначені для користувача каталоги.

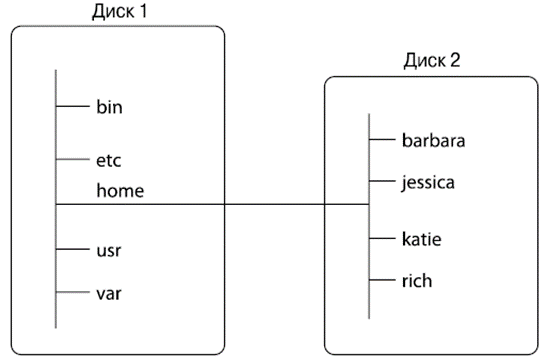


Рис. 1.3. Файлова архітектура системи Linux

Структура файлової системи Linux була розроблена на основі файлової архітектури UNIX. На жаль, файлова архітектура UNIX за минулі роки в зв'язку з появою різних різновидів UNIX втратила колишню однаковість. Складається таке враження, ніби не можна знайти такі дві системи UNIX або Linux, в яких застосовувалася б повністю однакова структура файлової системи. Проте передбачено кілька загальноприйнятих імен каталогів, які призначені для виконання приблизно одних і тих же функцій. У табл. 1.1 перераховані деякі з найбільш широко застосовуваних імен підкаталогів віртуального каталогу Linux.

Табл. 1.1. Загальноприйняті імена каталогів Linux

|  |  |
| --- | --- |
| **Каталог** | **Призначення** |
| **/** | Корінь віртуального каталогу. Зазвичай прийнято не розміщувати в корені будь-які файли |
| **/bin** | Каталог двійкових файлів, в якому зберігається велика кількість програм для користувача рівня, що відносяться до категорії утиліт GNU |
| **/boot** | Завантажувальний каталог, в якому зберігаються завантажувальні файли |
| **/dev** | Каталог пристроїв, в якому Linux створює спеціальні файли пристроїв |
| **/etc** | Каталог файлів конфігурації системи |
| **/home** | Вихідний каталог, в якому Linux створює користувальницькі каталоги |
| **/lib** | Бібліотечний каталог, в якому зберігаються файли бібліотек системи і додатків |
| **/media** | Мультимедійний каталог - загальне місце для точок монтування, використовуваних для змінних носіїв |
| **/mnt** | Каталог монтування - ще одне загальноприйняте місце для точок монтування, використовуваних для змінних носіїв |
| **/opt** | Додатковий каталог, який часто служить для зберігання додаткових пакетів програм |
| **/root** | Домашній каталог користувача root |
| **/sbin** | Системний каталог двійкових файлів, в якому зберігається багато адміністративних програм на рівні GNU |
| **/tmp** | Тимчасовий каталог, в якому можуть створюватися і знищуватися тимчасові робочі файли |
| **/usr** | Каталог програм, встановлених користувачем |
| **/var** | Каталог зі змінним вмістом, призначений для файлів, які часто змінюються, таких як файли журналів |

Після запуску нового запрошення командного інтерпретатора відкривається сеанс з переходом у вихідний каталог користувача, який представляє собою унікальний каталог, який призначений для роботи тільки з певним обліковим записом користувача. При створенні облікового запису користувача система зазвичай призначає унікальний каталог виключно для цього облікового запису.

**Перехід по каталогах**

У файлової системи Linux для переходу в інший каталог в сеансі командного інтерпретатора використовується команда зміни каталогу **cd** (change directory). Формат команди **cd** є відносно нескладним:

**cd destination**

Команда **cd** може приймати єдиний параметр призначення, **destination**, який вказує ім'я каталогу, в який необхідно перейти. Якщо в команді **cd** не вказано призначення, то відбувається перехід у домашній каталог користувача.

Але параметр призначення може бути виражений з використанням двох різних способів:

* у вигляді абсолютного шляху до файлу;
* як відносний шлях до файлу.

**Абсолютний шлях до файлу**

Абсолютний шлях до файлу точно визначає, де знаходиться каталог в структурі віртуального каталогу, починаючи від кореня віртуального каталогу і закінчуючи ім'ям самого каталогу, що призводить до отримання повного імені каталогу.

Знаючи абсолютний шлях до файлу, можна не сумніватися в тому, що пошук об'єкта, на який він вказує, буде успішним. Щоб перейти в конкретне місце розташування в файлової системі з використанням абсолютного шляху до файлу, достатньо вказати це повне ім'я шляху в команді **cd**:

**$ cd /etc**

**Відносні шляхи до файлу**

Відносні шляхи до файлу дозволяють вказувати шлях до необхідного файлу щодо поточного місця розташування в файлової системі, не змушуючи починати з кореня. Позначення відносного шляху до файлу починається не з косою риси, яка вказує на кореневий каталог.

Замість цього позначення відносного шляху до файлу починається або з імені каталогу (якщо відбувається перехід до каталогу, розташованому нижче поточного каталогу), або зі спеціального символу, яке зазначає місце розташування щодо поточного каталогу. Для цього використовуються два спеціальні символи:

* точка (.), яка позначає поточний каталог;
* подвійна точка (..), яка представляє батьківський каталог.

Символ подвійний точки стає надзвичайно зручним, якщо потрібно виконати перехід по ієрархії каталогів. Наприклад, якщо поточним є підкаталог Documents вихідного каталогу і потрібно перейти в каталог Desktop, який також розташований нижче вихідного каталогу, це завдання можна вирішити наступним чином:

**$ cd ../Desktop**

Символ подвійної точки забезпечує перехід вгору на один рівень в структурі вихідного каталогу, а потім частина відносного шляху **/Desktop** дозволяє знову перейти на більш низький рівень, але вже в каталог Desktop. Для переміщення по структурі каталогів можна стільки раз використовувати символ подвійної точки, скільки буде потрібно. Наприклад, якщо необхідно перейти в каталог /etc з поточного каталогу, яким є домашній каталог, то можна ввести наступне:

**$ cd ../../etc**

Зрозуміло, в подібних випадках фактично доводиться застосовувати більший обсяг введення, щоб скористатися відносним шляхом до файлу, в порівнянні з безпосереднім зазначенням абсолютного шляху до файлу.

Однією з найбільш важливих функцій командного інтерпретатора є надання можливості отримання відомостей про те, які файли є в системі. Інструментом, що дозволяє досягти цієї мети, є команда **ls**(скорочення від list - список).

Команда **ls**, що застосовується в найбільш простій формі, відображає файли і каталоги, що знаходяться у поточному каталозі:

**$ ls**

**4rich Desktop Download Music Pictures store store.zip test**

**backup Documents Drivers myprog Public store.sql Templates Videos**

Заслуговує на увагу те, що команда **ls**формує лістинг в алфавітному порядку (це стосується стовпців, а не рядків). Якщо для роботи використовується емулятор терміналу, який підтримує колір, то за допомогою команди **ls**можна також сформувати виведення, в якому записи різних типів позначені різними кольорами. Цим засобом керує змінна середовища **LS\_COLORS**. У різних дистрибутивах Linux ця змінна середовища задається в залежності від можливостей емулятора терміналу.

У табл. 1.2 перерахована частина найбільш широко застосовуваних параметрів, за допомогою яких можна отримати необхідні дані, працюючи з командою ls командного інтерпретатора bash.

Табл 1.2. Деякі широко застосовуються параметри команди **ls**

|  |  |  |
| --- | --- | --- |
| **Однобуквений** | **Що складається з повних слів** | **Опис** |
| **-a** | **--all** | Чи не пропускати записи, що починаються з точки |
| **-d** | **--directory** | Виводити в лістинг записи з позначеннями каталогів замість їх вмісту і не розадресовувати символічні посилання |
| **-F** | **--classify** | Приєднувати до записів позначення типів файлів |
| **-h** | **--human-readable** | Виводити дані про розміри з використанням позначення K для кілобайт (1024 байти), M для мегабайтів і G для гігабайтів |
| **-i** | **--inode** | Відображати індексний номер (так званий індексний вузол) для кожного файлу |
| **-l** |  | Сформувати лістинг в довгому форматі |
| **-r** | **--reverse** | Застосовувати зворотний порядок сортування при відображенні файлів і каталогів |
| **-s** | **--size** | Виводити розмір кожного файлу в блоках |
| **-S** | **--sort = size** | Сортувати висновок за розмірами файлів |
| **-t** | **--sort = time** | Сортувати висновок за часом модифікації файлів |

Параметр **-F**, що дозволяє сформувати виведення, в якому простіше відрізнити файли від каталогів. Застосування параметра -F призводить до отримання наступного виведення:

**$ Ls -F**

**4rich/ Documents/ Music/ Publi / store.zip Videos/**

**backup.zip Download/ myprog\* store/ Templates/**

**Desktop/ Drivers/ Pictures/ store.sql test**

У виведенні, сформованому за допомогою параметра **-F**, каталоги позначені косою рисою, що дозволяє простіше знаходити їх в лістингу. Крім того, даний параметр забезпечує застосування зірочки для позначення виконуваних файлів.

Проте застосування команди **ls**в основній формі може призвести до деякої плутанини. Вона показує файли і каталоги, що містяться в поточному каталозі, але не обов'язково всі. В системі Linux часто використовуються приховані файли для зберігання інформації про конфігурацію. Приховані файли в системі Linux - це файли з іменами файлів, що починаються з точки. Такі файли не з'являються в створюваному за замовчуванням лістингу **ls**(саме тому вони називаються прихованими). Для відображення прихованих файлів поряд зі звичайними файлами і каталогами використовується параметр **-a**.

Може також застосовуватися ще один важливий параметр команди **ls**- параметр **-R**, який дозволяє дізнатися, які файли знаходяться в підкаталогах поточного каталогу. Якщо кількість підкаталогів досить велике, то лістинг, що формується із застосуванням зазначеного параметра, може виявитися вельми довгим. Нижче наведено простий приклад результатів, отриманих при використанні параметра **-R**.

**$ Ls -F -R**

**.:**

**file1 test1 / test2 /**

**./test1:**

**myprog1 \* myprog2 \***

**./test2:**

Для отримання в лістингу додаткових відомостей служить ще один широко відомий параметр, **-l**. Параметр **-l**(скорочення від long - довгий) дозволяє отримати так званий довгий формат лістингу, в якому наведена докладна інформація про кожен файл в каталозі:

**$ Ls -l**

**total 2064**

**drwxrwxr-x 2 rich rich 4096 2010-08-24 22:04 4rich**

**-rw-r - r-- 1 rich rich 1766205 2010-08-24 15:34 backup.zip**

**drwxr-xr-x 3 rich rich 4096 2010-08-31 22:24 Desktop**

**drwxr-xr-x 2 rich rich 4096 2009-11-01 4:06 Documents**

**drwxr-xr-x 2 rich rich 4096 2009-11-01 4:06 Download**

**drwxrwxr-x 2 rich rich 4096 2010-07-26 18:25 Drivers**

**drwxr-xr-x 2 rich rich 4096 2009-11-01 4:06 Music**

**-rwxr - r-- 1 rich rich 30 2010-08-23 21:42 myprog**

**drwxr-xr-x 2 rich rich 4096 2009-11-01 4:06 Pictures**

**drwxr-xr-x 2 rich rich 4096 2009-11-01 4:06 Public**

**drwxrwxr-x 5 rich rich 4096 2010-08-24 22:04 store**

**-rw-rw-r-- 1 rich rich 98772 2010-08-24 15:30 store.sql**

**-rw-r - r-- 1 rich rich 107507 2010-08-13 15:45 store.zip**

**drwxr-xr-x 2 rich rich 4096 2009-11-01 4:06 Templates**

**drwxr-xr-x 2 rich rich 4096 2009-11-01 4:06 Videos**

У кожному рядку лістингу в довгому форматі містяться відомості про різні файлах і каталогах, наявних в даному каталозі. Такий лістинг, крім імені файлу, показує іншу корисну інформацію. У першому рядку виведення містяться відомості про загальну кількість блоків даних, що відносяться до поточного каталогу. Слідом за цим відбувається висновок окремих рядків, кожна з яких включає наступну інформацію про кожен файл (або каталозі).

* тип файлу, такий як каталог (**d**), файл (**-**), символьний пристрій (**c**) або блочний пристрій (**b**);
* дозволу для файлу;
* кількість жорстких посилань на файл;
* ім'я користувача власника файлу;
* ім'я групи файлів, до якої належить цей файл;
* розмір файлу в байтах;
* час останньої зміни файлу;
* ім'я файлу або каталогу.

**Обробка файлів та каталогів**

Командний інтерпретатор **bash** надає велику кількість команд маніпулювання файлами в файлової системі Linux. Далі наведено короткий опис основних команд, які застосовуються для роботи з файлами з інтерфейсу командного рядка у всіх ситуаціях, коли потрібно здійснювати обробку файлів.

**Створення файлів**

Час від часу виникають ситуації, в яких потрібно створити порожній файл. Наприклад, іноді функціональність таких програм засновано на тому, що в системі вже є файл журналу, в якому вони могли б виконувати операції запису. У подібних випадках при відсутності необхідного файлу можна використовувати команду touch, що дозволяє легко створити порожній файл:

**$ touch test1**

**$ ls -il test1**

**1954793 -rw-r - r-- 1 rich rich 0 Sep 1 9:35 test1**

Команда **touch**створює новий файл з вказаним ім'ям, а в якості власника файлу призначає ім'я поточного користувача. У розглянутій команді **ls**використовувався параметр **-il**, тому перший запис в лістингу показує номер індексного вузла, присвоєний файлу. Кожен файл у файловій системі Linux має унікальний номер індексного вузла.

**Копіювання файлів**

Команда cp копіює файли і каталоги. Її можна використовувати двома різними способами:

**cp item1 item2**

щоб скопіювати один файл або каталог item1 в файл або каталог item2, і

**cp елемент ... каталог**

щоб скопіювати декілька елементів (файлів або каталогів) в зазначений каталог.

Як і більшість інших команд, команда cp дозволяє скористатися численними параметрами командного рядка, які призначені для вирішення багатьох різних завдань. Ці параметри наведені в табл. 1.3.

Табл. 1.3. Параметри команди **cp**

|  |  |
| --- | --- |
| **параметр** | **опис** |
| **-a** | Запис файлів в архів зі збереженням їх атрибутів |
| **-i** | Висновок запрошення перед перезаписом цільових файлів |
| **-p** | Збереження атрибутів файлів, якщо це можливо |
| **-r** | Рекурсивне копіювання каталогів |
| **-u** | Копіювання вихідного файлу, тільки якщо він має більш нові значення дати і часу в порівнянні з цільовим файлом (проведення оновлення) |
| **-v** | Застосування режиму докладного виведення з поясненням того, що відбувається |

**приклади**

**cp file1 file2**- скопіює file1 в file2. Якщо file2 існує, він буде затертий новим файлом file1. Якщо file2 відсутня, він буде створений.

**cp -i file1 file2**- те саме, що і вище, але якщо файл file2 існує, у користувача буде запитано підтвердження перед перезаписом файлу.

**cp file1 file2 dir1**- скопіює file1 і file2 в каталог dir1. Каталог dir1 повинен існувати

**cp -r dir1 dir2**- скопіює каталог dir1 (і весь його вміст) в каталог dir2. Якщо каталог dir2 не існує, він буде створений і заповнений вмістом каталогу dir1.

**Переміщення і перейменування файлів**

Команда **mv**виконує операції переміщення і перейменування файлів в залежності від особливостей використання. У будь-якому випадку вихідний файл зникає після операції. Команда **mv**використовується майже так само, як команда **cp** :

**mv item1 item2 -**переміщує або перейменовує файл або каталог item1 в item2.

**mv елемент ... каталог**- переміщує один або більше елементів з одного каталогу в інший.

Команда **mv** підтримує безліч тих же параметрів, що і команда **cp** .

**приклади**

**mv file1 file2 - п**ереместить file1 в file2. Якщо file2 існує, він буде замінений на новий файл file1. Якщо file2 відсутня, він буде створений. У будь-якому випадку з'явиться новий файл file2;

**mv -i file1 file2 -**те саме, що і вище, але якщо файл file2 існує, у користувача буде запитано підтвердження перед перезаписом файлу;

**mv file1 file2 dir1**- перемістить file1 і file2 в каталог dir1. Каталог dir1 повинен існувати

**mv dir1 dir2 - п**ереместіть каталог dir1 (і весь його вміст) в каталог dir2. Якщо каталог dir2 не існує, він буде створений і заповнений вмістом каталогу dir1. Каталог dir1 буде видалений.

**Створення каталогів**

Команда **mkdir**використовується для створення каталогів. Викликається вона наступним чином:

**mkdir каталог ...**

**приклади**

**mkdir dir1 -**створить один каталог з ім'ям dir1;

**mkdir dir1 dir2 dir3**- створить три каталогу з іменами dir1, dir2 і dir3.

**Видалення файлів і каталогів**

Команда **rm**використовується для видалення (remove) файлів і каталогів, наприклад:

**rm елемент ...**

де **елемент**- це один або кілька файлів або каталогів.

Табл. 2.4. Параметри команди **rm**

|  |  |  |
| --- | --- | --- |
| **короткий** | **довгий** | **опис** |
| **-i** | **--interactive** | Запитувати у користувача підтвердження перед видаленням існуючого файлу. Якщо цей параметр відсутній, команда rm просто видалить існуючі файли |
| **-r** | **--recursive** | Рекурсивно видалити каталоги. Тобто разом з каталогом будуть видалені всі його підкаталоги. Це обов'язковий параметр при видаленні каталогів |
| **-f,** | **--force** | Ігнорувати відсутні файли і не запрошувати підтвердження. Цей параметр скасовує дію параметра --interactive |
| **-v** | **--verbose** | Виводити інформаційні повідомлення в процесі видалення |

**приклади**

**rm file1**- просто видалить файл file1;

**rm -i file1**- перед видаленням file1 запросить підтвердження у користувача;

**rm -r file1 dir1**- видалить файл file1 і каталог dir1 з усім його вмістом;

**rm -rf file1 dir1**- те саме, що і вище, але за відсутності file1 і / або dir1 просто продовжить роботу, не виводячи жодних повідомлень.

**Створення посилань**

Команда **ln**застосовується для створення жорсткого або символічного посилання. Її можна використовувати одним з двох способів:

**ln файл посилання -**створює жорстке посилання;

**ln -s елемент посилання -**створює символічне посилання, де елементом може бути файл або каталог.

**Жорсткі посилання**

Жорсткі посилання - це початковий спосіб створення посилань в UNIX; символічні посилання - пізніший винахід. За замовчуванням кожен файл має одне жорстке посилання, що визначатиме її ім'я. Створюючи жорстке посилання, ми створюємо додатковий запис в каталозі для файлу. Жорсткі посилання мають два важливих обмеження.

* Жорстке посилання не може вказувати на файл за межами власної файлової системи. Це означає, що посилання не може вказувати на файл, що знаходиться в іншому розділі диска.
* Жорстке посилання не може вказувати на каталог.

Жорстке посилання відрізняється від самого файлу. На відміну від символічних посилань, при виведенні списку з вмістом каталогів жорсткі посилання ніяк не виділяються. При видаленні жорсткої посилання видаляється тільки саме посилання, а файл залишається на місці (тобто простір, займаний файлом, не звільняється), поки не будуть видалені всі жорсткі посилання на файл.

Знати про існування жорстких посилань важливо, тому що вони будуть зустрічатися вам час від часу, але в сучасній практиці перевага віддається символічним посиланням, про яких розповідається далі.

**Символічні посилання**

Символічні посилання були придумані з метою подолати обмеження жорстких посилань. Коли створюється символічне посилання, насправді створюється файл особливого типу, що містить текстовий покажчик на файл або каталог. В деякому відношенні вони діють подібно ярликів в Windows, але, звичайно ж, з'явилися задовго до ярликів Windows.

Файл, на який вказує символьне посилання, і саме символічне посилання майже не відрізняються один від одного. Наприклад, якщо спробувати щось записати в символічне посилання, запис буде виконаний в файл, на який воно вказує. Однак при видаленні символічного посилання видаляється лише символічне посилання, але не файл. Якщо видалити файл до того, як буде видалена символічна посилання, посилання залишиться на місці, але буде вказувати в нікуди. Про таких посиланнях кажуть, що вони «биті». У багатьох реалізаціях команда **ls** виділяє биті посилання кольором, наприклад, червоним, щоб звернути на них увагу.

**Лекція №2**

**«Стандартні потоки даних у Linux та їх перенаправлення. Механізм конвеєрів та використання фільтрів»**

**Стандартні потоки даних**

Багато програм, якими ми вже користувалися, що-небудь виводять на консоль. Це виведення часто ділиться на два типи. Перший - результати роботи програми, тобто дані, для отримання яких створювалася програма. Другий - повідомлення про стан або про помилки, які сповіщають нас про самопочуття програми. Наприклад, якщо поглянути на виведення програми **ls**, можна побачити, що вона виводить на екран результати своєї роботи і іноді повідомлення про помилки.

Згідно центральної ідеї UNIX, що «все суще є файл», такі програми, як **ls**, насправді виводять свої результати в спеціальний файл, який називається **стандартним виведенням** (standard output, часто позначається як stdout), а повідомлення про стан - в спеціальний файл **стандартне виведення помилок** (standard error, stderr). За замовчуванням обидва файли, стандартне виведення і стандартне виведення помилок, пов'язані з екраном і не зберігаються на диску.

Крім того, багато програм приймають введення зі спеціального файлу з назвою **стандартне введення** (standard input, stdin), який за замовчуванням пов'язаний з клавіатурою.

Механізм перенаправлення вводу / виводу дозволяє змінювати напрямок виведення і введення. Зазвичай виведення здійснюється на екран, а введення - з клавіатури, але механізм перенаправлення введення/виведення дозволяє змінити цей порядок речей.

**Перенаправлення стандартного виведення**

Механізм перенаправлення введення/виведення дозволяє явно вказати, куди повинне здійснюватися стандартне висновок. Щоб перенаправити стандартне виведення в інший файл замість екрану, потрібно додати в команду оператор перенаправлення **>** і ім'я файлу. Де це може стати в нагоді? Іноді корисно зберегти виведення команди в файл. Наприклад, можна повідомити командній оболонці, що вона повинна направити виведення команди **ls** в файл **ls-output.txt** замість екрану:

**$ ls -l /usr/bin > ls-output.txt**

Тут ми створили довгий список вмісту файлу **/usr/bin** і відправили результати в файл **ls-output.txt**. Давайте дослідимо перенаправлене виведення команди:

**$ ls -l ls-output.txt**

**-rw-rw-r-- 1 me me 167878 2012-02-01 15:07 ls-output.txt**

Якщо вивести вміст **ls-output.txt**за допомогою команди **less**, можна побачити, що він дійсно містить результати роботи команди **ls**:

**$ less ls-output.txt**

Давайте тепер повторимо експеримент з перенаправленням, але з невеликим ускладненням: вкажемо ім'я неіснуючого каталогу:

**$ ls -l / bin / usr> ls-output.txt**

**ls: неможливо отримати доступ до '/bin/usr': Немає такого файлу або каталогу**

Ми отримали повідомлення про помилку. Як і багато програм для UNIX, команда **ls**виводить повідомлення про помилки в стандартний потік виведення помилок. Оскільки ми перенаправили тільки стандартне виведення, а стандартне виведення помилок - ні, повідомлення про помилку з'явилося на екрані. Як перенаправити стандартне виведення помилок, буде показано трохи нижче, але перед цим подивимося, що сталося з нашим файлом:

**$ ls -l ls-output.txt**

**-rw-rw-r-- 1 me me 0 2012-02-01 15:08 ls-output.txt**

Простий оператор перенаправлення, без попередньої йому команди, очистить існуючий файл або створить новий, порожній файл.

Щоб додати виведення в кінець існуючого файлa, не перезаписавши його, потрібно використовувати оператор перенаправлення **>>**:

**$ ls -l / usr / bin >> ls-output.txt**

Оператор **>>** просто додасть результати в кінець файлу. Якщо файл не існує, він буде створений, як при використанні оператора **>**.

**Перенаправлення стандартного виведення помилок**

Перенаправлення стандартного виведення помилок здійснюється не так просто, як стандартного виведення. Щоб перенаправити стандартне виведення помилок, потрібно вказати його дескриптор файлу. Програма може виробляти виведення у будь-який з кількох нумерованих файлових потоків. Перші три з них ми згадали як стандартне введення, виведення і виведення помилок. Командна оболонка посилається на них як на файлові дескриптори **0**, **1** і **2** відповідно. Командна оболонка підтримує синтаксис перенаправлення файлів з використанням номерів файлових дескрипторів. Так як стандартному виведенню помилок відповідає файловий дескриптор **2**, ми можемо перенаправити його, як показано нижче:

**$ ls -l /bin/usr 2> ls-error.txt**

Номер файлового дескриптора **2** поміщується безпосередньо перед оператором перенаправлення, щоб перенаправити стандартне виведення помилок в файл **ls-error.txt**.

**Перенаправлення стандартного виведення і стандартного виведення помилок в один файл**

Іноді необхідно зберегти все виведення команди в один файл. Для цього потрібно перенаправити відразу два потоки: стандартне виведення і стандартне виведення помилок. Зробити це можна двома способами. Перший - традиційний - працює в старих версіях командної оболонки:

**$ ls -l /bin/usr > ls-output.txt 2>&1**

Тут виконується два перенаправлення. Спочатку - перенаправлення стандартного виведення у файл **ls-output.txt**, а потім, з використанням нотації  **2>&1**, - перенаправлення файлового дескриптора **2** (стандартне виведення помилок) в файловий дескриптор 1 (стандартне виведення).

Сучасні версії **bash** підтримують другий, більш простий метод виконання перенаправлення цього виду:

**$ ls -l /bin/usr &> ls-output.txt**

В даному прикладі використовується єдиний оператор **&>**, що перенаправляє стандартне виведення і стандартне виведення помилок в файл **ls-output.txt**.

**Видалення небажаного виведення**

Система дає таку можливість, надаючи спеціальний файл **/dev/null**, куди можна перенаправити виведення. Цей файл являє системний пристрій, який називається **бітоприймачем** (bit bucket), або Кошика, яке приймає будь-введення і нічого з ним не робить. Щоб придушити висновок повідомлень про помилки, досить виконати наступне:

**$ ls -l /bin/usr 2> /dev/null**

**Використання команди cat для різних завдань**

Команда **cat**читає вміст одного або декількох файлів і копіює його в стандартний висновок:

**cat [файл ...]**

Часто команду  **cat**можна вважати аналогом команди TYPE в DOS. Вона використовується для виведення вмісту файлів без можливості посторінкового перегляду. наприклад,

**$ cat ls-output.txt**

виведе вміст файлу **ls-output.txt**. Команда **cat**часто використовується для виведення коротких текстових файлів. Оскільки **cat**здатна приймати відразу кілька файлів, вона використовується для їх об'єднання. Уявіть, що ви завантажили великий файл, розбитий на безліч частин (в Usenet мультимедійні файли часто розбиваються таким способом), і потрібно об'єднати їх в один файл. Якщо файли мають імена, такі як

**movie.mpeg.001 movie.mpeg.002 ... movie.mpeg.099**

їх можна об'єднати за допомогою такої команди:

**$ cat movie.mpeg.0 \*> movie.mpeg**

Оскільки підстановка фактичних імен замість групових символів завжди виконується в порядку сортування, аргументи виявляться розташованими в правильному порядку.

Якщо викликати **cat**без аргументів, вона почне читати дані зі стандартного введення, а оскільки стандартне введення за замовчуванням підключене до клавіатури, виходить, що команда чекає, поки ми що-небудь надрукуємо!

**$ cat**

**З'їж ще цих м'яких французьких булок, та випий чаю.**

Якщо потім натиснути комбінацію **CTRL-D** (тобто, утримуючи клавішу **CTRL**, натиснути клавішу **D**), щоб повідомити команді **cat**, що досягнутий кінець файлу (end-of-file, EOF) на стандартному введенні:

**$ cat**

**З'їж ще цих м'яких французьких булок, та випий чаю.**

**З'їж ще цих м'яких французьких булок, та випий чаю.**

За відсутності аргументів з іменами файлів **cat**копіює вміст стандартного введення в стандартне виведення, тому-то ми і побачили, як вона повторила введений нами рядок. Цю її особливість можна використовувати для створення коротких текстових файлів. Уявіть, що вам потрібно створити файл з ім'ям  **eat\_more.txt**, що містить текст із прикладу, наведеного вище. Зробити це можна було б так:

**$ cat> eat\_more.txt**

**З'їж ще цих м'яких французьких булок, та випий чаю.**

Щоб побачити результат, скористаємося командою **cat** і скопіюємо файл в стандартне виведення:

**$ cat eat\_more.txt**

**З'їж ще цих м'яких французьких булок, та випий чаю.**

Тепер, коли ми знаємо, що команда **cat** може приймати дані не тільки з файлів, зазначених в аргументах, а й зі стандартного введення, спробуємо виконати перенаправлення стандартного введення:

**$ cat <eat\_more.txt**

**З'їж ще цих м'яких французьких булок, та випий чаю.**

Використовуючи оператор перенаправлення **<**, ми змінили джерело даних для стандартного введення з клавіатури на файл **eat\_more.txt**. Як можна побачити, результат вийшов той самий, як якщо б ми просто передали єдиний аргумент з ім'ям файлу. Цей спосіб не має ніяких переваг в порівнянні з передачею простого аргументу, але він демонструє, як можна використовувати файли в ролі джерела даних для стандартного введення. Інші команди знаходять краще застосування стандартного введення.

**Групові символи**

Так як імена файлів використовуються в командній оболонці повсюдно, вона підтримує спеціальні символи, що допомагають швидко визначати групи імен файлів. Ці спеціальні символи називають **груповими символами** (wildcards). Групові символи (також відомі як **символи підстановки** (globbing)) дозволяють вибирати імена файлів за шаблоном. У табл. 2.1 перераховані групові символи і їх відповідності.

Табл. 2.1. Групові символи

|  |  |
| --- | --- |
| **груповий символ** | **Відповідає** |
| **\*** | Будь-яка послідовність будь-яких символів |
| **?** | Будь-який символ |
| **[Символи]** | Будь-який символ з вказаної множини символів |
| **[!Символи]** | Будь один символ, що не належить зазначеному безлічі символів |
| **[[:Клас:]]** | Будь один символ, що належить зазначеному класу |

У табл. 2.2 представлені найбільш часто використовувані класи символів.

Табл. 2.2. Найбільш часто використовувані класи символів

|  |  |
| --- | --- |
| **клас символів** | **Відповідає** |
| **[:Alnum:]** | Будь алфавітно-цифровий символ |
| **[:Alpha:]** | Будь-яка літера алфавіту |
| **[:Digit:]** | Будь-цифровий символ |
| **[:Lower:]** | Будь-яка буква в нижньому регістрі |
| **[:Upper:]** | Будь-яка буква у верхньому регістрі |

**Конвеєри**

Уміння команд читати дані зі стандартного введення і виводити результати в стандартне виведення використовується механізмом командної оболонки, який називається **конвеєром**. За допомогою оператора конвеєра **|**(вертикальна риса) стандартне виведення однієї команди можна пов'язати зі стандартним введенням іншої.

**команда1 | команда2**

**Фільтри**

Конвеєри часто використовуються для виконання складних операцій з даними. Вони дозволяють об'єднати разом кілька команд. Часто команди, використовувані таким способом, називають **фільтрами**. Фільтри приймають введення, змінюють його певним чином і виводять результат. Перший з таких фільтрів - команда **sort**. Наприклад, нам необхідно скласти список всіх виконуваних програм в каталогах **/bin**та **/usr/bin**, розташувавши їх за алфавітом, і потім вивести його:

**$ ls/bin/usr/bin | sort | less**

Оскільки в команді вказані два каталоги ( **/bin**і **/usr/bin**), виведення команди **ls**буде складатися з двох відсортованих списків, по одному для кожного каталогу. Додавши команду **sort**в конвеєр, ми змінили дані, щоб отримати єдиний сортований список.

**Uniq - пошук або видалення повторюваних рядків**

Команда **uniq**часто використовується в комбінації з командою **sort**. **uniq**приймає сортований список даних або зі стандартного введення, або з файлу, ім'я якого можна передати в єдиному аргументі, і за замовчуванням видаляє повторювані рядки зі списку. Тому, щоб гарантувати відсутність дублікатів в нашому списку (тобто будь-яких програм з однаковими іменами в каталогах **/bin**та **/usr/bin**), додамо **uniq**в конвеєр:

**$ ls /bin/usr/bin | sort | uniq | less**

У цьому прикладі ми використовували **uniq**для видалення будь-яких повторюваних рядків у виведенні команди **sort**. Якби нам треба було, навпаки, отримати список дублікатів, ми додали б в команду **uniq** параметр **-d**:

**$ ls /bin/usr/bin | sort | uniq -d | less**

**wc - виведення числа рядків, слів і байтів**

Команда **wc**(word count - лічильник слів) використовується для підрахунку числа рядків, слів і байтів в файлах. наприклад:

**$ wc ls-output.txt**

**7902 64566 503634 ls-output.txt**

В даному випадку команда вивела три числа: число рядків, число слів і число байтів у файлі **ls-output.txt**. Подібно попереднім командам, вона може викликатися без аргументів, і в цьому випадку **wc**буде приймати дані зі стандартного введення. Параметр **-l**обмежує виведення результатів тільки числом рядків. Команду зручно використовувати в конвеєрах для підрахунку: наприклад, підрахувати число елементів в нашому сортованому списку можна так:

**$ ls /bin/usr /bin | sort | uniq | wc -l**

**2728**

**grep - пошук рядків, відповідних шаблоном**

**grep**- дуже потужна програма, вона часто використовується для пошуку в файлах тексту за шаблоном:

**grep шаблон [файл ...]**

Коли **grep**знаходить в файлі збіг з «шаблоном», вона виводить рядок зі знайденим збігом. Шаблони, що використовуються командою **grep**для пошуку, можуть бути дуже складними, але зараз ми розглянемо тільки пошук прямого збігу з текстом. Припустимо, що нам потрібно знайти всі файли в списку програм, які мають в своєму імені послідовність символів **zip**. Результати такого пошуку можуть підказати нам, які програми в системі мають відношення до стиснення файлів. Зробити це можна так:

**$ ls /bin/usr /bin | sort | uniq | grep zip**

**bunzip2**

**bzip2**

**gunzip**

**gzip**

**unzip**

**zip**

**zipcloak**

**zipgrep**

**zipinfo**

**zipnote**

**zipsplit**

Команда **grep**має пару зручних параметрів:

**-i**вимагає від **grep** ігнорувати регістр символів в процесі пошуку (зазвичай пошук виконується з урахуванням регістра символів),

**-v**вимагає від **grep** виводити лише рядки, де збіг з шаблоном, не знайдено.

**head/tail - виведення перших/останніх рядків з файлів**

Іноді потрібно виводити не всі результати роботи команди, а тільки декілька перших або декілька останніх рядків. Команда **head**виводить перші 10 рядків з файлу, а **tail**- останні 10 рядків. За замовчуванням обидві команди виводять 10 рядків тексту, але це число можна змінити за допомогою параметра **-n**:

**$ head -n 5 ls-output.txt**

**total 343496**

**-rwxr-xr-x 1 root root 31316 2011-12-05 8:58 [**

**-rwxr-xr-x 1 root root 8240 2011-12-09 13:39 411toppm**

**-rwxr-xr-x 1 root root 111276 2011-11-26 14:27 a2p**

**-rwxr-xr-x 1 root root 25368 2010-10-06 20:16 a52dec**

**[Me @ linuxbox ~] $ tail -n 5 ls-output.txt**

**-rwxr-xr-x 1 root root 5234 2011-06-27 10:56 znew**

**-rwxr-xr-x 1 root root 691 2009-09-10 4:21 zonetab2pot.py**

**-rw-r - r-- 1 root root 930 2011-11-01 12:23 zonetab2pot.pyc**

**-rw-r - r-- 1 root root 930 2011-11-01 12:23 zonetab2pot.pyo**

**lrwxrwxrwx 1 root root 6 2012-01-31 5:22 zsoelim -> soelim**

Їх також можна використовувати в конвеєрах:

**$ ls /usr /bin | tail -n 5**

**znew**

**zonetab2pot.py**

**zonetab2pot.pyc**

**zonetab2pot.pyo**

**zsoelim**

Команда **tail**дозволяє спостерігати, як змінюється вміст файлу в режимі реального часу. Цю її особливість зручно використовувати для спостереження за появою нових записів в файлах журналів. У наступному прикладі демонструється спостереження за файлом **messages**в каталозі **/var/log**. У деяких дистрибутивах Linux для цього потрібно володіти привілеями суперкористувача, оскільки файл **/var/log/messages**може містити секретну інформацію.

**$ tail -f /var/log/messages**

**Feb 8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1**

**………………………………………………………………………………………………………………………………………………………………………….**

**Feb 8 14:25:36 twin4 su (pam\_unix) [29279]: session opened for user root by me (uid = 500)**

Коли ми викликаємо з параметром **-f**команда **tail**продовжує стежити за файлом і при додаванні в кінець цього файлу нових рядків негайно виводить їх. Так триває до тих пір, поки користувач не натисне комбінацію клавіш **CTRL-C**.

**tee - читання зі стандартного вводу і запис у стандартне виведення і у файли**

Linux надає команду **tee**, яка створює Т-образне розгалуження в конвеєрі. Програма **tee** читає дані зі стандартного введення і копіює їх у стандартне виведення (щоб дати можливість передати їх далі по конвеєру) і в один або кілька файлів. Це може стати в нагоді для збереження проміжних результатів обробки в конвеєрі. Нижче, продовжуючи один з попередніх прикладів, ми збережемо повний список файлів в каталогах у файлі **ls.txt**, перед тим як він буде відфільтрований командою **grep**:

**$ ls /usr/bin | tee ls.txt | grep zip**

**bunzip2**

**bzip2**

**gunzip**

**gzip**

**unzip**

**Текстові редактори**

Для редагування (тобто зміни) файлів запуску командної оболонки, а також більшості інших конфігураційних файлів в системі використовується програма, яка називається **текстовим редактором.** Текстовий редактор, подібно текстовому процесору, дозволяє редагувати слова на екрані, переміщаючи курсор. Від текстового процесора ця програма відрізняється тільки підтримкою простого тексту і нерідко наявністю особливостей, необхідних при розробці програм. Текстові редактори - основний інструмент, який використовується програмістами для створення програмного коду і системними адміністраторами для керування файлами, що визначають налаштування системи.

Для Linux існує величезна кількість текстових редакторів. Чому було створено так багато редакторів? Ймовірно, тому, що програмістам подобається писати їх, а так як програмісти дуже активно користуються редакторами, вони прагнуть втілити в них свої погляди на те, як повинні працювати ці редактори.

Текстові редактори діляться на дві основні категорії: з графічним і з текстовим інтерфейсом. Обидва оточення робочого столу, **GNOME** і **KDE**, включають кілька популярних редакторів з графічним інтерфейсом. До складу GNOME входить редактор з назвою **gedit**, який в меню GNOME зазвичай називається **Text Editor** (Текстовий редактор). Разом з KDE зазвичай поширюється три редактора (в порядку збільшення складності): **kedit**, **kwrite**і **kate**.

Існує безліч редакторів з текстовим інтерфейсом. Найбільш популярні з них **nano**, **vi** і **emacs**. Редактор **nano**- простий у використанні редактор, створений як заміна редактору **pico**, що поставляється в складі пакета програм для роботи з електронною поштою PINE. Редактор **vi**(в більшості систем Linux його заміщає програма **vim**, назва якої є скороченням від Vi IMproved (Vi покращений)) - традиційний редактор для Unix-подібних систем. Редактор **emacs**був написаний Річардом Столлманом. Це неймовірне, універсальне середовище програмування, побудоване за принципом «все в одному». Але, не дивлячись на свою доступність, він рідко встановлюється за умовчанням в більшості систем Linux.

**Використання текстового редактора nano**

Будь-який текстовий редактор можна запустити з командного рядка, увівши ім'я редактора і ім'я файлу, який потрібно відредагувати. Якщо вказаний файл не існує, редактор вирішить, що ви хочете створити новий файл. Нижче наводиться приклад використання **gedit**:

**$ gedit some\_file**

Ця команда запустить текстовий редактор **gedit** і завантажить в нього файл з ім'ям **some\_file**, якщо такий існує.

Всі текстові редактори з графічним інтерфейсом мають інтуїтивно зрозумілий інтерфейс, тому ми не будемо описувати їх тут. Замість цього зосередимося на редакторі з текстовим інтерфейсом **nano**. Давайте запустимо **nano**і внесемо зміни в файл **.bashrc**.

**$ nano .bashrc**

Після запуску nano ми побачимо на екрані приблизно таку картину (рис. 2.1):



Рис. 2.1. Зовнішній вигляд текстового редактора **nano**

Екран редактора ділиться на три частини: заголовок у верхній частині, область редагування тексту в середині і меню команд знизу. Так як **nano**проектувався як заміна текстового редактора, що входить до складу поштового клієнта, він не володіє розвиненими функціями редагування.

Перша команда, яку потрібно дізнатися при використанні будь-якого редактора, - це команда виходу з програми. Щоб залишити **nano**, необхідно скористатися комбінацією клавіш **CTRL+X**. Ця команда присутня в меню, в нижній частині екрана. Нотація **^X**означає **CTRL+X**. Це поширена форма запису керуючих комбінацій, що використовувалися в багатьох програмах. Друга команда, яку слід знати, - як зберегти зміни. У **nano** збереження виконується натисканням **CTRL+O**.

**Текстовий редактор vi**

Навіщо в сучасному світі редакторів з графічним інтерфейсом і простих у використанні редакторів з текстовим інтерфейсом, таких як **nano**, освоювати **vi**?

1. **vi**завжди під рукою. Він може прийти на допомогу в системах, де відсутній графічний інтерфейс, наприклад на віддаленому сервері або в локальної системі з неробочий конфігурацією X. Редактор **nano**, хоча і надзвичайно популярний, все ж недостатньо універсальний .POSIX, стандарт програмної сумісності систем Unix, вимагає наявності в них vi.
2. **vi**легкий і швидкий . для багатьох завдань набагато простіше запустити vi, ніж знайти в меню редактор з графічним інтерфейсом і чекати, поки кілька мегабайтів завантажиться в пам'ять. Крім того, vi спеціально проектувався для швидкісного введення з клавіатури.

Перша версія **vi**була написана Біллі Джоем, студентом Каліфорнійського університету в місті Берклі, який пізніше став співзасновником Sun Microsystems в 1976 році. Назва vi походить від слова visual (екранний), тому що редактор призначався для редагування на екрані відеотермінала з можливістю переміщення курсору по всій його поверхні. До екранних редакторів існували малі редактори, що дозволяють редагувати текст тільки по одному рядку. Щоб внести зміни в рядковому редакторі, потрібно було спочатку перейти до необхідного рядка а потім описати необхідну зміну: додавання або видалення тексту. З появою відеотерміналів (замість терміналів з друкуючим пристроєм, таких як телетайпи) стало можливим візуальне редагування на екрані. Насправді vi включає в себе потужний рядковий редактор ex, і його можна використовувати для введення команд під час роботи в **vi**.

Більшість дистрибутивів Linux містять не справжній редактор **vi**, а його поліпшену заміну з ім'ям **vim**(скорочено від Vi IMproved - Vi покращений), створену Брамом Моленаром (Bram Moolenaar). **vim**істотно досконаліше традиційного редактора **vi**і в системах Linux зазвичай використовується під символічною посиланням (або псевдонімом) **vi**.

**Запуск і завершення vi**

Щоб запустити **vi**, потрібно ввести наступну команду:

**$ vi**

На екрані має з'явитися:

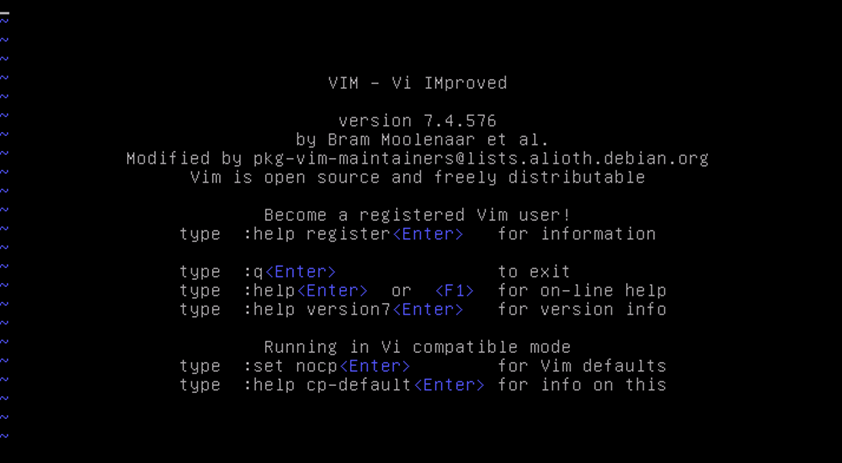


Рис. 2.2. Зовнішній вигляд текстового редактора vi (vim)

Для виходу з редактора потрібно ввести наступну команду (двокрапка - це частина команди):

**:q**

У вікні терміналу має з'явитися запрошення до введення командної оболонки. Якщо з якоїсь причини вихід з **vi**не вийшов, потрібно повідомити **vi**, що ми дійсно хочемо вийти, додавши в команду знак оклику:

**:q!**

**Режими редагування**

Знову запустимо **vi**, але на цей раз зазначимо ім'я неіснуючого файлу. Саме так можна за допомогою **vi**створити новий файл:

**$ rm -f foo.txt**

**$ vi foo.txt**

Початкові символи тильди (~) повідомляють про відсутність тексту у відповідних рядках. Таким способом **vi**повідомляє нам, що файл порожній.

Друга важлива річ, яку потрібно засвоїти (після того, як ви навчилися виходити), - **vi**є режимним редактором. Відразу після запуску **vi**виявляється в **командному режимі** . В цьому режимі практично кожна клавіша є командою.

Щоб додати якийсь текст в файл, необхідно спочатку перейти в **режим вставки**. Для цього потрібно натиснути клавішу **I(i)**.

Щоб вийти з **режиму вставки** і повернутися в **командний режим**, потрібно натиснути **ESC**.

**Збереження змін**

Щоб зберегти зміни в файл, необхідно ввести ex-команду, перебуваючи в командному режимі. Для цього потрібно натиснути клавішу**:** . Після цього в нижній частині повинен з'явитися символ двокрапки:

**:**

Щоб виконати запис змін в файл, слідом за двокрапкою потрібно ввести **w** і натиснути **ENTER** :

**: w**

Файл буде записаний на жорсткий диск, і в нижній частині з'явиться підтвердження:

**"Foo.txt" [New] 1L, 46C written**

**Переміщення курсора**

Перебуваючи в командному режимі, **vi**пропонує велике число команд управління курсором, частина з яких також використовується програмою **less**. У табл. 2.3 перераховані деякі з цих команд.

Табл. 2.3. Клавіші управління курсором

|  |  |
| --- | --- |
| **клавіша** | **переміщує курсор** |
| ***L або стрілка вправо*** | Вправо на один символ |
| ***H або стрілка вліво*** | Вліво на один символ |
| ***J або стрілка вниз*** | Вниз на один рядок |
| ***K або стрілка вгору*** | Вгору на один рядок |
| ***0 (нуль)*** | На початок поточного рядка |
| ***SHIFT+6 (^)*** | До першого непробільного символу в поточному рядку |
| ***SHIFT+4 ($)*** | В кінець поточного рядка |
| ***W (w)*** | На початок наступного слова або до знаку пунктуації |
| ***SHIFT+w (W)*** | На початок наступного слова, минаючи розділові знаки |
| ***B (b)*** | На початок попереднього слова або до знаку пунктуації |
| ***SHIFT+b (B)*** | На початок попереднього слова, минаючи розділові знаки |
| ***CTRL+F або Page Down*** | Вниз на одну сторінку |
| ***CTRL+B або Page Up*** | Вгору на одну сторінку |
| ***число-SHIFT+g*** | До рядку з зазначеним номером (наприклад, команда 1G виконає перехід до першого рядка у файлі) |
| ***SHIFT+g (G)*** | До останнього рядка у файлі |

Чому для переміщення курсора були обрані клавіші **H**, **J**, **K**і **L**? Тому що, коли був написаний редактор **vi**, не всі відеотермінали мали кнопки зі стрілками на клавіатурі. Таким чином, досвідчені користувачі, що добре володіють клавіатурою, могли управляти курсором, не відриваючи пальців від клавіш.

Багато команд в **vi**можуть починатися з числа, як команда **G**в табл. 4.1. Додаючи число в команду, можна вказати, скільки разів вона повинна бути виконана. Наприклад, команда **5j**перемістить курсор на п'ять рядків вниз.

**Основи редагування у vi**

Редагування в основному полягає в декількох простих операціях, таких як вставка тексту, видалення тексту і переміщення фрагментів тексту із застосуванням операцій вирізання та вставки. Звичайно ж, **vi**підтримує всі ці операції своїм неповторним способом. **vi**підтримує обмежену форму скасування. Якщо натиснути клавішу **U**в командному режимі, **vi**скасує останню виконану зміну.

**Додавання тексту в кінець**

**vi**підтримує кілька способів входу в режим вставки, крім **i**.

Якщо спробувати додати текст в кінець програми, можна виявити, що команда **i**не дозволяє зробити це, не даючи перемістити курсор за кінець рядка. **vi**підтримує команду додавання тексту в кінець, розумно названу **a**. Якщо перемістити курсор в кінець рядка і ввести **a**, курсор переміститься за кінець рядка і **vi** перейде в режим вставки.

Оскільки додавати текст в кінець рядка потрібно досить часто, **vi**пропонує скорочену команду для переміщення в кінець рядка і переходу в режим додавання. Це команда **A**.

**Вставка рядка**

Інший спосіб вставки тексту - вставка рядків. Він дозволяє вставити порожній рядок між двома наявними рядками і перейти в режим вставки. Даний спосіб пропонує два варіанти вставки:

**o**- рядок нижче поточного;

**O**- рядок вище поточного.

**Видалення тексту**

**vi**пропонує кілька способів видалення тексту, і всі вони вимагають натискання однієї або двох клавіш. Перший спосіб: клавіша **x**видаляє символ в позиції курсора. Команді **x**може передувати число, що визначає кількість символів, що видаляються. Кнопка **d**більш універсальна. Команді **d**також може передувати число, що визначає кількість операцій видалення. Крім того, команда **d**завжди супроводжується командою переміщення курсору, що управляє розміром видаляється області. У табл. 2.4 наводиться кілька прикладів команди видалення.

Табл. 2.4. Команди видалення тексту

|  |  |
| --- | --- |
| **команда** | **видаляє** |
| **X** | Поточний символ |
| **3x** | Поточний символ і наступні за ним два символи |
| **dd** | Поточний рядок |
| **5dd** | Поточний рядок і наступні за ним чотири рядки |
| **dW** | Від символу в поточній позиції курсору до початку наступного слова |
| **d$** | Від символу в поточній позиції курсору до кінця поточного рядка |
| **d0** | Від символу в поточній позиції курсору до початку рядка |
| **d^** | Від символу в поточній позиції курсору до першого непробільного символу в рядку |
| **dG** | Від поточного рядка до кінця файлу |
| **d20G** | Від поточного рядка до 20-го рядка файлу |

**Вирізання, копіювання і вставка тексту**

Команди, що починаються з символу **d,**не просто видаляють текст, вони «вирізають» його. Кожен раз, коли виконується команда, що починається з символу **d**, віддалений текст копіюється в буфер вставки (свого роду буфер обміну - clipboard), звідки пізніше його можна витягти командою **p**і вставити правіше позиції курсора або лівіше - командою **P**.

Команда **y**виконує копіювання (yank) тексту в буфер вставки майже так само, як команда **d**. У табл. 2.5 перераховані деякі приклади комбінування команди **y**з різними командами переміщення курсора.

Табл. 2.5. Команди копіювання тексту

|  |  |
| --- | --- |
| **команда** | **Скопіює** |
| **yy** | Поточний рядок |
| **5yy** | Поточний і наступні чотири рядки |
| **yW** | Від поточної позиції курсора до початку наступного слова |
| **y $** | Від поточної позиції курсора до кінця поточного рядка |
| **y0** | Від поточної позиції курсора до початку рядка |
| **y ^** | Від поточної позиції курсора до першого непробельний символу в рядку |
| **yG** | Від поточного рядка до кінця файлу |
| **y20G** | Від поточного рядка до 20-го рядка файлу |

**Об'єднання рядків**

**vi**дуже суворо ставиться до поняття рядка. Зазвичай він не дає можливості перемістити курсор в кінець рядка і видалити символ кінця рядка, щоб об'єднати поточний рядок з наступною за нею. З цієї причини в vi була додана спеціальна команда **J**(не плутати з командою **j**, яка переміщує курсор на один рядок вниз) для об'єднання рядків.

**Пошук і заміна у vi**

Редактор **vi**має можливість переміщати курсор, спираючись на результати пошуку. Він може це робити в межах одного рядка або всього файлу. Він може також виконувати заміну тексту з підтвердженням або без підтвердження користувача.

**Пошук у межах рядка**

Команда **f**виконує пошук в рядку і переміщує курсор до наступного входження зазначеного символу. Наприклад, команда **fa**перемістить курсор до наступного входження символу **a** в поточному рядку. Після виконання операції пошуку символу в рядку її можна повторити, ввівши крапку з комою.

**Пошук у всьому файлі**

Для переміщення курсору до наступного входження слова або фрази використовується команда **/**. Вона діє так само, як в програмі **less**. Після введення команди **/**в нижній частині екрана з'явиться прямий слеш, слідом за яким потрібно ввести шукане слово або фразу і натиснути ENTER. Після цього курсор переміститься до наступного входження шуканої рядка. Пошук наступного входження тієї ж рядки можна повторити командою **n**.

**Глобальний пошук і заміна**

Для виконання пошуку з заміною (в **vi**ця операція називається підстановкою) в діапазоні рядків або у всьому файлі vi використовує ex-команди. Наприклад, замінити слово **Рядок**словом **рядок**у всьому файлі можна за допомогою такої команди:

**:%S/Рядок/рядок/g**

Розіб'ємо цю команду на елементи і розглянемо їх окремо (табл. 2.6).

Табл. 2.6. Приклад синтаксису команди глобального пошуку з заміною

|  |  |
| --- | --- |
| **елемент** | **значення** |
| **:** | Символ двокрапки починає ex-команду |
| **%** | Визначає діапазон рядків, де буде виконуватися пошук. % - скорочення, що означає «від першого рядка до останнього». У цьому прикладі можна було б вказати діапазон номерів рядків 1,5 (тому що в нашому файлі всього п'ять рядків) або 1, $, що означає «від рядка з номером 1 до останньої». Якщо діапазон рядків не вказано, операція застосовується тільки до поточного рядка |
| **s** | Визначає операцію - в даному випадку підстановку (substitution) або пошук з заміною |
| **/Line/line/** | Шаблон пошуку і текст заміни |
| **g** | Означає global (глобально), в тому сенсі, що підстановка виконується для всіх входжень шуканої рядка в кожному рядку. Якщо елемент g опустити, операція виконає заміну тільки першого входження шуканого фрагмента в кожному рядку |

У команді підстановки можна вказати, що вона повинна запитувати підтвердження у користувача перед заміною. Для цього потрібно додати символ **c**в кінець команди. наприклад:

**:%S/рядок/Рядок/gc**

Ця команда поверне вміст файлу в попередній стан, але перед кожною заміною **vi**буде зупинятися і питати підтвердження, видаючи наступне повідомлення:

**замінити на Рядок? (Y/n/a/q/l/^E/^Y)**

В круглих дужках перераховані можливі варіанти відповідей, опис яких наводиться в табл. 2.7.

Табл. 2.7. Клавіші підтвердження заміни

|  |  |
| --- | --- |
| **клавіша** | **Дія** |
| **y** | виконати заміну |
| **n** | Пропустити знайдене входження |
| **a** | Виконати заміну цього і всіх наступних входжень |
| **q**або **ESC** | завершити операцію |
| **l** | Виконати заміну цього входження і завершити операцію. Скорочено від last (остання) |
| **CTRL + E, CTRL + Y** | Прокрутити вниз або вгору відповідно. Ці команди зручно використовувати для перегляду контексту знайденого входження перед заміною |

**Збереження результатів роботи у vi**

Тут **vi**пропонує нам кілька способів збереження відредагованих файлів. Ми вже знайомі з ex-командою **:w**, але існують і інші команди, які можуть виявитися корисними.

У командному режимі можна ввести **ZZ**, щоб зберегти поточний файл і вийти з **vi**. Аналогічно, ex-команда **:wq**поєднує в собі команди **:w**і **:q**, перша з яких зберігає файл, а друга закриває редактор. У команді **:w**можна також вказати ім'я файлу. В цьому випадку вона буде діяти як команда Save As (Зберегти як).

**Лекція №3**

**«Права доступу для файлів та каталогів Linux, їх встановлення. Права доступу за замовчуванням. Основні команди роботи з текстовими файлами»**

Операційні системи, що слідують традиціям UNIX, відрізняються від систем, тащо слідують традиціям MS-DOS, тим, що є не тільки багатозадачними, але й багатокористувацькими.

Що це означає насправді? Це означає, що комп'ютером можуть одночасно користуватися кілька людей. Незважаючи на те що зазвичай комп'ютер має всього одну клавіатуру і монітор, ця обставина не заважає спільному користуванню. Наприклад, якщо комп'ютер підключений до локальної мережі або до Інтернету, віддалені користувачі зможуть зайти на нього через **ssh** (secure shell - безпечна командна оболонка) і виконувати операції. Фактично віддалені користувачі можуть запускати додатки з графічним інтерфейсом і отримувати зображення на віддаленому дисплеї. X Window System підтримує таку можливість спочатку.

**Власники, члени групи і всі інші**

У моделі безпеки UNIX користувач може володіти файлами і каталогами. Якщо користувач володіє файлом або каталогом, він може керувати доступом до нього. Користувачі можуть також належати групі, що складається з одного або декількох користувачів, і отримати права доступу до файлів і каталогів для членів групи, які визначаються власниками. Крім прав доступу для групи, власник може також визначити деякі права доступу для всіх інших, їх в термінології UNIX називають **світ (world)** . Отримати інформацію про свою ідентичність можна за допомогою команди **id**:

**$ id**

**uid = 500 (me) gid = 500 (me) groups = 500 (me)**

Розглянемо це виведення. Коли створюється обліковий запис користувача, йому присвоюється число, яке називають **ідентифікатором користувача (user ID)**, або **uid** . Це число, виключно заради зручності людини, відображається як ім'я користувача. Користувачеві призначається **ідентифікатор основної групи (primary group ID)**, або **gid**, і додатково користувач може включатися до складу інших груп. Попередній приклад взятий з системи Fedora. В інших системах, таких як Ubuntu, виведення команди може трохи відрізнятися.

**$ id**

**uid = 1000 (me) gid = 1000 (me)**

**groups = 4 (adm), 20 (dialout), 24 (cdrom), 25 (floppy), 29 (audio), 30 (dip), 44 (video), 46 (plugdev), 108 (lpadmin), 114 (admin), 1000 (me)**

Числа **uid** і **gid** відрізняються. Це пояснюється тим, що в Fedora нумерація облікових записів звичайних користувачів починається з 500, тоді як в Ubuntu - з 1000. Крім того, користувач в Ubuntu належить безлічі інших груп. Це пов'язано з особливостями управління привілеями доступу до системних пристроїв і службам в Ubuntu.

А де ж вся ця інформація зберігається? Як і багато іншого в Linux, вона зберігається в парі текстових файлів. Облікові записи користувачів зберігаються в файлі **/etc/passwd**, а інформація про групи - в файлі **/etc/group**. Коли створюються нові облікові записи і групи, ці файли змінюються разом з файлом **/etc/shadow**, де зберігається інформація про пароль користувача. Для кожного облікового запису в файлі **/etc/passwd**визначається ім'я користувача (для входу), числовий ідентифікатор користувача (**uid** ), числовий ідентифікатор основної групи ( **gid** ), дійсне ім'я користувача, шлях до домашнього каталогу і командна оболонка входу ( **login shell** ). Зазирнувши всередину **/etc/passwd**і **/etc/group**, можна помітити, що крім облікових записів звичайних користувачів тут також зберігаються облікові записи суперкористувача (uid 0) і різних інших системних користувачів.

Незважаючи на те що в багатьох Unix-подібних системах звичайних користувачів включають в загальну групу, таку як **users**, в сучасних дистрибутивах Linux прийнято створювати для кожного користувача свою, унікальну групу з одним членом і ім'ям, що збігається з ім'ям користувача. Це спрощує розподіл певних типів привілеїв.

**Права доступу до файлів і каталогів**

Права доступу до файлів і каталогів визначаються в термінах права **на читання**, права **на запис** і права **на виконання** . Якщо поглянути на висновок команди **ls**, можна побачити деякі підказки про те, як ці права реалізовані:

**$ > foo.txt**

**$ ls -l foo.txt**

**-rw-rw-r-- 1 me me 0 2012-03-06 14:52 foo.txt**

Перші 10 символів у висновку - це атрибути файлу (рис. 3.1).

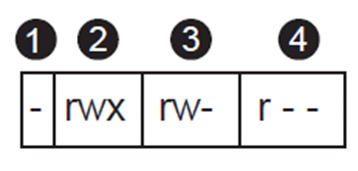


Рис. 3.1. атрибути файлу

**1 - Тип файлу; 2 - Привілеї для власника; 3 - Привілеї для групи; 4 - Привілеї для всіх інших.**

Табл. 3.1. типи файлів

|  |  |
| --- | --- |
| **Атрибут** | **Тип файлу** |
| **-** | звичайний файл |
| **d** | Каталог |
| **l** | Символічне посилання. Для символічних посилань всі інші атрибути мають значення **rwxrwxrwx**і не відображають дійсні права доступу. Фактичні права доступу до файлу визначаються атрибутами самого файлу, на який вказує символьне посилання |
| **c** | Спеціальний файл символьного пристрою. Файли цього типу відповідають пристроям, таким як термінал або модем, які обробляють дані як потоки байтів |
| **b** | Спеціальний файл блочного пристрою. Файли цього типу відповідають пристроям, таким як привід жорсткого диска або CD-ROM, які обробляють дані блоками |

Табл. 3.2. Атрибути прав доступу

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **файли** | **Каталоги** |
| **r** | Дозволяється відкривати і читати вміст файлу | Дозволяється читати вміст каталогу, якщо разом з цим атрибутом встановлений атрибут права на виконання |
| **w** | Дозволяється записувати в файл або усікати його; однак цей атрибут не дає права перейменовувати і видаляти файли. Можливість перейменування і видалення файлів визначається атрибутами каталогу, що вмішує дані файли | Дозволяється створювати, видаляти і перейменовувати файли усередині каталогу, якщо разом з цим атрибутом встановлений атрибут права на виконання |
| **x** | Дозволяється інтерпретувати файл як програму і виконувати її. Файли, що містять програми на мовах сценаріїв, додатково повинні бути доступні для читання, інакше вони не будуть виконуватися | Дозволяється входити в каталог, тобто виконувати команду **cd** для переходу в нього |

У табл. 3.3 наводиться кілька прикладів встановлення атрибутів файлів.

Табл. 3.3. Приклади встановлення атрибутів прав доступу до файлів

|  |  |
| --- | --- |
| **атрибути файлів** | **значення** |
| **-rwx ------** | Звичайний файл, доступний власнику для читання, запису і виконання. Ніхто інший не має прав доступу до файлу |
| **-rw -------** | Звичайний файл, доступний власнику для читання і запису. Ніхто інший не має прав доступу до файлу |
| **-rw-r - r--** | Звичайний файл, доступний власнику для читання і запису. Члени групи мають право читати файл. Всі інші мають право читати файл |
| **-rwxr-xr-x** | Звичайний файл, доступний власнику для читання, запису і виконання. Всі інші мають право читати і виконувати файл |
| **-rw-rw ----** | Звичайний файл, доступний для читання і запису тільки власнику і членам групи |
| **Lrwxrwxrwx** | Символічне посилання. Все символічні посилання мають недійсні значення атрибутів. Фактичні права доступу до файлу визначаються атрибутами самого файлу, на який вказує символьне посилання |
| **drwxrwx ---** | Каталог. Власник і члени групи можуть входити в каталог, створювати, перейменовувати і видаляти файли всередині каталогу |
| **drwxr-x ---** | Каталог. Власник може входити в каталог, створювати, перейменовувати і видаляти файли всередині каталогу. Члени групи можуть входити в каталог, але не можуть створювати, перейменовувати і видаляти файли всередині каталогу |

**Зміна режиму доступу до файлу**

Для зміни режиму (прав) доступу до файлу або каталогу використовується команда **chmod**. Майте на увазі, що права доступу до файлу або каталогу може змінити тільки власник. Команда **chmod**підтримує два різні способи зміни режиму: з використанням вісімкових чисел і символічного представлення.

**Вісімкове представлення**

При використанні вісімкової форми запису шаблон бажаних привілеїв визначається вісімковими числами. Так як кожна цифра в вісімковому числі визначається трьома двійковими розрядами, вона точно відображається в схему зберігання режиму доступу до файлу.

Табл. 3.4. Режими доступу до файлу в двійковому і вісімковому представленнях

|  |  |  |
| --- | --- | --- |
| **вісімкове** | **двійкове** | **режим доступу** |
| **0** | **000** | **---** |
| **1** | **001** | **--x** |
| **2** | **010** | **-w-** |
| **3** | **011** | **-wx** |
| **4** | **100** | **r--** |
| **5** | **101** | **rx** |
| **6** | **110** | **rw-** |
| **7** | **111** | **rwx** |

C допомогою трьох вісімкових цифр ми можемо визначити режим доступу до файлу для власника, для групи і для решти світу.

**$ > foo.txt**

**$ ls -l foo.txt**

**-rw-rw-r-- 1 me me 0 2012-03-06 14:52 foo.txt**

**$ chmod 600 foo.txt**

**$ ls -l foo.txt**

**-rw ------- 1 me me 0 2012-03-06 14:52 foo.txt**

Передавши аргумент **600**, ми встановили права для власника, що дозволяють йому читати дані з файлу і записувати їх в файл, і при цьому відібрали всі права у групи і решти світу. Незважаючи на удавану незручність необхідності запам'ятовування відповідностей між вісімковими і двійковими уявленнями, вам, швидше за все, доведеться використовувати лише кілька найбільш популярних шаблонів: **7 (rwx)**, **6 (rw-)**, **5 (rx)**, **4 (r--)**і **0 (---)**.

**Символічне представлення**

Команда **chmod**підтримує також символічну форму визначення режимів доступу до файлу. Символічна форма запису ділиться на три частини:

1. для кого встановлюються дозволу;
2. які операції з дозволами будуть виконуватися ;
3. на які дозволи ці операції будуть впливати.

Щоб вказати, для кого встановлюються дозволу, використовується комбінація символів u, g, o і a, як показано в табл. 3.5.

Табл. 3.5. Символічна форма запису аргументів команди **chmod**

|  |  |
| --- | --- |
| **символ** | **значення** |
| **u** | Скорочено від **user** (користувач), означає власника файлу або каталогу |
| **g** | Група |
| **o** | Скорочено від **other** (інші, інші), означає весь інший світ |
| **a** | Скорочено від **all** (всі); комбінація з усіх трьох символів: **u**, **g**і **o** |

Якщо не вказано ні один символ, передбачається **a**(all - всі). Операцією може бути:

* знак **+**, відповідний додаванню заданих дозволів;
* знак -, відповідними ющий відбирання заданих дозволів;
* знак =, який вказує, що тільки задані дозволу повинні бути встановлені, а все решта відібрані.

Дозволи визначаються символами **r**, **w**і **x**. У табл. 3.6 перераховані деякі приклади символічної форми запису.

Табл. 3.6. Приклади символічної форми записи прав доступу до файлів

|  |  |
| --- | --- |
| **атрибути файлів** | **значення** |
| **u+x** | Додає право на виконання, але тільки для власника |
| **u-x** | Забирає право на виконання у власника |
| **+x** | Додає право на виконання для власника, групи і решти світу. Еквівалент записи **a+x** |
| **o-rw** | Забирає право на читання і запис у всіх, крім власника і групи |
| **go=rw** | Встановлює право на читання і запис для всіх, крім власника. Якщо раніше файл мав дозвіл на виконання для групи і всього світу, це право віднімається |
| **u+x, go=rx** | Додає право на виконання для власника і встановлює право на читання і виконання для групи і всього світу. При виконанні відразу декількох операцій з привілеями вони повинні розділятися комою. |

Символічна форма запису зручна тим, що дозволяє встановити єдиний атрибут, не впливаючи на інші.

**Визначення дозволів доступу до файлів за замовчуванням**

Команда **umask**визначає дозвіл за замовчуванням, які встановлюються для файлу при його створенні. У ній за допомогою вісімкової форми запису визначається бітова маска для скидаються атрибутів режиму доступу.

Наприклад:

**$ rm -f foo.txt**

**$ umask**

**0002**

**$ > foo.txt**

**$ ls -l foo.txt**

**-rw-rw-r-- 1 me me 0 2012-03-06 14:53 foo.txt**

Спочатку ми видалили існуючий файл **foo.txt**, щоб, так би мовити, почати з чистого аркуша. Далі ми виконали команду **umask**без аргументу, щоб побачити поточне значення маски. Вона повернула нам значення **0002**(часто також використовується значення **0022**) - вісімкове представлення діючої маски. Потім ми створили новий файл **foo.txt**і вивели для нього дозвіл.

Власник і група отримали права на читання і запис, тоді як всі інші - тільки право на читання. Весь світ не отримав права на запис через значення маски. Давайте повторимо приклад, але на цей раз визначимо свою маску:

**$ rm foo.txt**

**$ umask 0000**

**$ > foo.txt**

**$ Ls -l foo.txt**

**-rw-rw-rw- 1 me me 0 2012-03-06 14:58 foo.txt**

Після встановлення маски в значення **0000**(таким способом ми фактично вимкнули її) новостворений файл отримав дозвіл на запис для всього світу. Щоб краще зрозуміти суть того, що відбувається, ми знову повинні повернутися до вісімкових чисел. Якщо розгорнути маску в двійкове представлення і порівняти її з двійковим представленням атрибутів, можна зрозуміти, що сталося:

|  |  |
| --- | --- |
| **Вихідний режим доступу до файлу** | --- rw- rw- rw- |
| **маска** | 000 000 000 010 |
| **результат** | --- rw- rw- r-- |

Атрибут, що відповідає тій позиції, де в масці стоїть 1, був скинутий, - в даному випадку право на запис для всього світу. Тепер зрозуміло, що робить маска. У будь-якій позиції, де в масці з'являється 1, відповідний атрибут скидається. Якщо подивитися на значення маски 0022, легко побачити, що воно робить:

|  |  |
| --- | --- |
| **Вихідний режим доступу до файлу** | --- rw- rw- rw- |
| **маска** | 000 000 010 010 |
| **результат** | --- rw- r-- r-- |

**Зміна власника та групи файлу**

Команда **chown**використовується для зміни власника і групи файлу або каталогу. Для використання цієї команди необхідні повноваження супер. Команда **chown**має наступний синтаксис:

**chown [власник] [: [група]] файл ...**

**chown**може змінити власника і / або групу файлу в залежності від першого аргументу. У табл. 5.7 наводиться кілька прикладів команди.

Табл. 3.7. Приклади аргументів команди **chown**

|  |  |
| --- | --- |
| **аргумент** | **результати** |
| **bob** | Змінити приналежність файлу, призначивши власником користувача **bob** |
| **bob: users** | Змінити приналежність файлу, призначивши власником користувача **bob**і групу **users** |
| **: admins** | Змінити приналежність файлу, призначивши групу **admins** |
| **bob:** | Змінити приналежність файлу, призначивши власником користувача **bob**і групу цього користувача |

Уявімо, що існують два користувача: **janet** , що має доступ до привілеїв суперкористувача, і **tony**, позбавлений таких привілеїв. Користувачеві **janet** потрібно скопіювати файл зі свого домашнього каталогу в домашній каталог користувача **tony**. Оскільки користувач **janet** хоче, щоб користувач **tony** зміг редагувати файл, **janet** повинен змінити власника скопійованого файлу, призначивши власником **tony** :

**$ sudo cp myfile.txt ~ tony**

**Password:**

**$ sudo ls -l ~ tony / myfile.txt**

**-rw-r - r-- 1 root root 8031 ​​2012-03-20 14:30 /home/tony/myfile.txt**

**$ sudo chown tony: ~ tony / myfile.txt**

**$ sudo ls -l ~ tony / myfile.txt**

**-rw-r - r-- 1 tony tony 8031 ​​2012-03-20 14:30 /home/tony/myfile.txt**

Тут видно, як користувач **janet** копіює файл зі свого каталогу в домашній каталог користувача **tony**.  Далі **janet** замінює власника файлу **root** (результат використання **sudo**) на **tony** . Додавши двокрапку в кінець першого аргументу, **janet** одночасно змінює відповідну групу, наприклад файл, на основну групу користувача **tony** , яка, так вже вийшло, носить те ж ім'я **tony** .

У більшості конфігурацій **sudo** продовжує «довіряти» користувачеві протягом декількох хвилин (поки не закінчиться час її дії).

У старих версіях UNIX команда **chown**змінювала тільки власника файлу, але не групу. Щоб змінити групу, надавалася інша команда, **chgrp**. Вона діє практично так само, як **chown**, але має більше обмежень.

**Команди для роботи з текстом**

Всі Unix-подібні операційні системи широко використовують текстові файли для зберігання даних різних типів. Цим пояснюється така велика різноманітність інструментів обробки тексту. У даній лекції ми розглянемо програми, які використовуються для виконання самих різних маніпуляцій з текстом.

Раніше були розглянуті деякі команди, здатні приймати дані не тільки з аргументів командного рядка, але і зі стандартного введення. Тоді ми дуже коротко познайомилися з ними, але зараз прийшов час ближчого знайомства - ми дізнаємося, як їх можна використовувати для обробки тексту.

**Об'єднання файлів і виведення їх в стандартний потік виведення**

Програма **cat**містить безліч цікавих параметрів. Багато з них використовуються для поліпшеного відображення текстового вмісту. Прикладом може служити параметр **-A**, який використовується для відображення недрукованих символів в тексті. Іноді необхідно знати, чи є керуючі символи в тексті, що проглядається. Найбільш поширеними з них є символи табуляції (на противагу проміжків) і символи повернення каретки, часто представляють кінці рядків в текстових файлах, оформлених в стилі MS-DOS. Іншим поширеним варіантом є файл, який містить рядки з завершальними пробілами.

Давайте створимо файл для експериментів, використовуючи **cat**як примітивний текстовий процесор. Для цього введемо команду **cat**(вказавши файл для перенаправлення виводу), а слідом наш текст, завершивши рядки натисканням клавіші ENTER і закінчивши всі комбінацією **CTRL+D**- вона повідомить програмі **cat**, що досягнутий кінець файлу. У цьому прикладі ми ввели символ табуляції і додали в кінець рядка кілька прогалин:

**$ cat > foo.txt**

**The quick brown fox jumped over the lazy dog.**

**$**

Далі, викличемо **cat**з параметром **-A**, щоб показати текст:

**$ cat -A foo.txt**

**^ IThe quick brown fox jumped over the lazy dog. $**

**$**

Як бачимо, символ табуляції в тексті представлений парою символів **^I**. Ця звичайна форма записи означає «**CTRL+I**», тобто, як виявляється, - символ табуляції. Тут також видно, що символ **$** зазначає істинний кінець рядка, допомагаючи побачити додаткові проміжки в кінці рядка.

Програма **cat**має також параметри, які використовуються для зміни тексту. Найбільш відомими є **-n**, який додає номери рядків, і **-s**, що пригнічує висновок безлічі порожніх рядків, що йдуть підряд. Давайте продемонструємо їх:

**$ cat > foo.txt**

**The quick brown fox**

**jumped over the lazy dog.**

**$ Cat -ns foo.txt**

**1 The quick brown fox**

**2**

**3 jumped over the lazy dog.**

**$**

У цьому прикладі ми створили нову версію тестового файлу **foo.txt**, що містить два рядки, розділені двома порожніми рядками. Після обробки тексту командою **cat**з параметрами **-ns**один порожній рядок була видалена, а інші рядки пронумеровані.

**Сортування рядків текстових файлів**

Програма **sort**сортує вміст стандартного введення або одного або декількох файлів, зазначених у командному рядку, і записує результати в стандартне виведення. Застосувавши той же прийом, який ми використовували спільно з командою **cat**, можна продемонструвати обробку стандартного введення.

**$ sort > foo.txt**

**c**

**b**

**a**

**$ cat foo.txt**

**a**

**b**

**c**

Після запуску команди ми ввели літери **c**, **b**і **a**, а потім ознаку кінця файлу за допомогою комбінації **CTRL+D**. Потім переглянули файл, і побачили, що рядки в ньому відсортовані.

Оскільки **sort**може приймати кілька файлів в аргументах командного рядка, існує можливість об'єднати безліч файлів в один відсортований файл. Наприклад, якщо у вас є три файли і вам потрібно об'єднати їх в один відсортований файл, це можна виконати наступним чином:

**$ sort file1.txt file2.txt file3.txt> final\_sorted\_list.txt**

Програма **sort**має кілька цікавих параметрів. Їх неповний список наводиться в табл. 3.8.

Табл. 3.8. Часто використовувані параметри команди **sort**

|  |  |  |
| --- | --- | --- |
| **параметр** | **довгий параметр** | **опис** |
| **-b** | **--ignore-leading-blanks** | За замовчуванням сортування виконується з урахуванням вмісту всього рядка, починаючи з першого символу. Цей параметр змушує sort ігнорувати початкові проміжки в рядках і при сортуванні починати порівняння рядків з перших непробільних символів |
| **-f** | **--ignore-case** | Виконувати сортування без урахування регістру символів |
| **-n** | **--numeric-sort** | Виконувати сортування, спираючись на числові значення рядків. Цей параметр дозволяє здійснювати сортування відповідно до числовими значеннями, а не за алфавітом |
| **-r** | **--reverse** | Сортувати в зворотньому порядку. Рядки в результаті будуть слідувати в порядку убування, а не зростання |
| **-k** | **--key = поле1 [, поле2]** | Сортувати по ключових полях, починаючи з поля1 і закінчуючи полем2, а не по всій рядку |
| **-m** | **--merge** | Інтерпретувати кожен аргумент як ім'я попередньо відсортованого файлу. Дозволяє об'єднати декілька файлів в загальний результат без виконання додаткової сортування |
| **-o** | **--output = файл** | Записати результат сортування не в стандартний висновок, а в зазначений файл |
| **-t** | **--field-separator = символ** | Визначає символ, роздільник полів. За замовчуванням як роздільник використовуються прогалини і символи табуляції |

Програма **sort**часто втягується в обробку табличних даних, таких як результат команди **ls**наприклад. Візьмемо файл з історією випуску нових версій трьох основних систем Linux в період з 2006 по 2008 рік. Кожен рядок у файлі містить три поля: назва дистрибутива, номер версії та дата випускати форматі ММ / ДД / РРРР:

**SUSE 10.2 12/07/2006**

**Fedora 10 11/25/2008**

**SUSE 11.0 06/19/2008**

**Ubuntu 8.04 04/24/2008**

**Fedora 8 11/08/2007**

**SUSE 10.3 10/04/2007**

**Ubuntu 6.10 10/26/2006**

**Fedora 7 05/31/2007**

**Ubuntu 7.10 10/18/2007**

**Ubuntu 7.04 04/19/2007**

**SUSE 10.1 05/11/2006**

**Fedora 6 10/24/2006**

**Fedora 9 05/13/2008**

**Ubuntu 6.06 06/01/2006**

**Ubuntu 8.10 10/30/2008**

**Fedora 5 03/20/2006**

Далі спробуємо впорядкувати файл і подивимося, що з цього вийде:

**$ Sort distros.txt**

**Fedora 10 11/25/2008**

**Fedora 5 03/20/2006**

**Fedora 6 10/24/2006**

**Fedora 7 05/31/2007**

**Fedora 8 11/08/2007**

**Fedora 9 05/13/2008**

**SUSE 10.1 05/11/2006**

**SUSE 10.2 12/07/2006**

**SUSE 10.3 10/04/2007**

**SUSE 11.0 06/19/2008**

**Ubuntu 6.06 06/01/2006**

**Ubuntu 6.10 10/26/2006**

**Ubuntu 7.04 04/19/2007**

**Ubuntu 7.10 10/18/2007**

**Ubuntu 8.04 04/24/2008**

**Ubuntu 8.10 10/30/2008**

У нас це майже вийшло. Єдина проблема виникла з сортуванням номерів версій Fedora. Так як в лексикографічному сенсі 1 передує 5, версія 10 виявилася вгорі, тоді як версії 9 - внизу.

Щоб виправити цю помилку, виконаємо сортування по декількох ключах. Отже, нам потрібно виконати сортування по першому полю в алфавітному порядку, а потім по другому полю в числовому порядку. Програма sort дозволяє вказати в командному рядку кілька параметрів **-k**, щоб можна було визначити кілька ключів сортування. Насправді в ключ можна включати діапазон полів. Якщо діапазон не визначений (як в прикладах, наведених вище), **sort**використовує в якості ключа частина рядка, що починається з вказаного поля і тягнеться до кінця рядка.

Ось як виглядає синтаксис сортування по декількох ключах:

**$ Sort --key=1,1 --key=2n distros.txt**

**Fedora 5 03/20/2006**

**Fedora 6 10/24/2006**

**Fedora 7 05/31/2007**

**Fedora 8 11/08/2007**

**Fedora 9 05/13/2008**

**Fedora 10 11/25/2008**

**SUSE 10.1 05/11/2006**

**SUSE 10.2 12/07/2006**

**SUSE 10.3 10/04/2007**

**SUSE 11.0 06/19/2008**

**Ubuntu 6.06 06/01/2006**

**Ubuntu 6.10 10/26/2006**

**Ubuntu 7.04 04/19/2007**

**Ubuntu 7.10 10/18/2007**

**Ubuntu 8.04 04/24/2008**

**Ubuntu 8.10 10/30/2008**

В аргументі для першого примірника параметра ключа ми вказали діапазон полів, що входять в перший ключ. Так як сортування повинна виконуватися тільки по першому полю, ми вказали діапазон **1,1**, що означає: «починаючи з поля 1 і закінчуючи полем 1». Другому примірнику ми передали аргумент **2n**, який означає: «ключем сортування є друге поле і сортування виконується в порядку числових значень». В кінці визначення ключа допускаються однобуквені імена параметрів, вони вказують на тип сортування. Імена цих однобуквених параметрів збігаються з іменами глобальних параметрів програми **sort**: **b**(ігнорувати початкові пробіли), **n**(числове сортування), r (сортування в зворотному порядку) і т. д.

Третє поле в списку містить дату в форматі, незручному для сортування. У комп'ютері дати зазвичай наводяться до виду РРРР-ММ-ДД, що спрощує сортування в хронологічному порядку, але тут використовується американський формат ММ/ДД/РРРР. Як же тоді впорядкувати цей список в хронологічному порядку?

На щастя, **sort**надає таку можливість. Параметр **--key**дозволяє визначати зміщення всередині полів, щоб в якості ключів можна було використовувати частини полів:

**$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt**

**Fedora 10 11/25/2008**

**Ubuntu 8.10 10/30/2008**

**SUSE 11.0 06/19/2008**

**Fedora 9 05/13/2008**

**Ubuntu 8.04 04/24/2008**

**Fedora 8 11/08/2007**

**Ubuntu 7.10 10/18/2007**

**SUSE 10.3 10/04/2007**

**Fedora 7 05/31/2007**

**Ubuntu 7.04 04/19/2007**

**SUSE 10.2 12/07/2006**

**Ubuntu 6.10 10/26/2006**

**Fedora 6 10/24/2006**

**Ubuntu 6.06 06/01/2006**

**SUSE 10.1 05/11/2006**

**Fedora 5 03/20/2006**

Додавши параметр **-k 3.7**, ми повідомили програмі **sort**, що вона повинна використовувати для сортування ключ, що починається з сьомого символу в третьому полі, який відповідає початку року. Аналогічно, параметри **-k 3.1**і **-k 3.4**визначають ключі сортування по місяцю і дня місяця. Ми також додали параметри **n**і **r**, щоб виконати числову сортування в зворотному порядку. Параметр **b**доданий для виключення початкових пробілів з поля з датою (число яких в різних рядках відрізняється і тим самим впливає на результат сортування).

У деяких файлах як роздільники використовуються символи, відмінні від проміжків і символів табуляції; візьмемо, наприклад, файл **/etc/passwd**.  Поля в цьому файлі відокремлюються один від одного символом двокрапки (:). Чи можна впорядкувати вміст цього файлу з використанням ключових полів? Програма **sort**підтримує параметр **-t**для визначення символу-роздільника полів. Щоб впорядкувати вміст файлу **passwd**по сьомому полю (командна оболонка за замовчуванням), використовуємо таку команду:

**$ Sort -t ':' -k 7 /etc/passwd | head**

**me:x:1001:1001:Myself,,,:/home/me:/bin/bash**

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

**dhcp:x:101:102::/nonexistent:/bin/false**

**gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false**

**hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false**

**klog:x:103:104::/home/klog:/bin/false**

**messagebus:x:108:119::/var/run/dbus:/bin/false**

**polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false**

**pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false**

**Виявлення або видалення повторюваних рядків**

У порівнянні з **sort**програма **uniq**більш легковажна. Вона вирішує, здавалося б, тривіальне завдання. Коли їй передається сортований файл (в тому числі і стандартне введення), вона видаляє повторювані рядки і виводить результат в стандартне виведення. Вона часто використовується в поєднанні з **sort**для видалення повторюваних рядків.

Давайте створимо текстовий файл для подальших експериментів:

**$ cat > foo.txt**

**a**

**b**

**c**

**a**

**b**

**c**

Не забудемо ввести **CTRL+D**, щоб завершити введення з клавіатури. Якщо тепер застосувати **uniq** до нашого текстового файлу, результат нічим не буде відрізнятися від оригіналу; повторювані записи нікуди не зникли:

**$ uniq foo.txt**

**a**

**b**

**c**

**a**

**b**

**c**

Щоб **uniq**дійсно виконала свою роботу, вихідні дані потрібно спочатку впорядкувати:

**$ sort foo.txt | uniq**

**a**

**b**

**c**

Це пояснюється тим, що **uniq**видаляє повторювані записи, тільки якщо вони слідують один за одним. uniq має кілька параметрів. Найбільш часто використовувані з них перераховані в табл. 6.2.

У наступному прикладі використовується параметр **-c**програми **uniq**для визначення числа повторюваних рядків у вихідному текстовому файлі:

**$ sort foo.txt | uniq -c**

**2 a**

**2 b**

**2 c**

Табл. 3.9. Часто використовувані параметри команди **uniq**

|  |  |
| --- | --- |
| **параметр** | **опис** |
| **-c** | Вивести список повторюваних рядків, випереджаючи їх числом знайдених дублікатів |
| **-d** | Вивести тільки повторювані, що не унікальні рядки |
| **-f n** | Пропустити n початкових полів в кожному рядку. Розподіл на поля проводиться по пробільним символам, як в програмі sort; проте, на відміну від sort, програма uniq не має параметра для налаштування альтернативного роздільника полів |
| **-i** | Порівнювати рядки без урахування регістру символів |
| **-s n** | Пропустити n початкових символів в кожному рядку |
| **-u** | Вивести тільки унікальні рядки. Мається на увазі за замовчуванням |

**Нарізка та перетасування тексту**

**Видалення фрагментів з усіх рядків у файлах**

Програма **cut**використовується для вилучення фрагментів тексту з рядків і виведення їх в стандартне виведення. Вона може приймати імена файлів в аргументах або дані зі стандартного введення.

Визначення фрагментів рядків, які підлягають вилученню, реалізовано не дуже зручно, і для цієї мети застосовуються параметри, перераховані в табл. 3.8.

Табл. 3.8. Параметри команди **cut** для вибору фрагментів

|  |  |
| --- | --- |
| **параметр** | **опис** |
| **-c спісок\_сімволов** | Витягує фрагмент рядка, який визначається списком символів. Список може включати один або кілька числових діапазонів, розділених комами |
| **-f спісок\_полей** | Витягує одне або кілька полів з рядка, як визначено списком\_символів. Список може включати одне або кілька полів або діапазонів полів, розділених комами |
| **-d сімвол\_разделітель** | У присутності параметра **-f**як роздільник полів використовується символ\_роздільник. За замовчуванням поля повинні відділятися один від одного одним символом табуляції |
| **--complement** | Витягує рядок тексту цілком, крім фрагментів, які визначаються параметром **-c**і/або **-f** |

Якщо скористатися програмою **cat**з параметром **-A**, можна побачити, чи відповідає файл вимогу щодо використання символу табуляції як роздільник полів.

**$ cut -f 3 distros.txt**

**12/07/2006**

**11/25/2008**

**06/19/2008**

**04/24/2008**

**11/08/2007**

10/04/2007

**10/26/2006**

**05/31/2007**

**10/18/2007**

**04/19/2007**

**05/11/2006**

**10/24/2006**

**05/13/2008**

**06/01/2006**

**10/30/2008**

**03/20/2006**

Так як поля в файлі **distros.txt**розділені символами табуляції, їх зручніше витягувати за допомогою **cut**саме як поля, а не як групи символів. Коли поля розділяються символами табуляції, малоймовірно, що рядки будуть містити одне і те ж число символів, через що визначення позицій символів в рядках стає складною або нерозв'язним завданням. У прикладі, наведеному вище, ми змогли витягти поля з датами, які, до нашої удачі, всі мають однакову довжину, тому тепер ми можемо показати, як виконується витяг груп символів, для чого спробуємо витягти рік з кожного рядка:

**$ cut -f 3 distros.txt | cut -c 7-10**

**2006**

**2008**

**2008**

**2008**

**2007**

**2007**

**2006**

**2007**

**2007**

**2007**

**2006**

**2006**

**2008**

**2006**

**2008**

**2006**

Застосувавши **cut**вдруге до нашого списку, ми змогли витягти символи в позиціях з 7-ї по 10-ю, які відповідають році в поле з датою. Форма запису **7-10** - це приклад визначення діапазону. Повний опис особливостей визначення діапазонів знаходиться на сторінці довідкового керівництва ( **man**) для команди **cut** .

При роботі з полями визначимо роздільник, що відрізняється від символу табуляції. Наступний приклад витягує перше поле з файлу **/etc/passwd**:

**$ cut -d ':' -f 1 /etc/passwd | head**

**root**

**daemon**

**bin**

**sys**

**sync**

**games**

**man**

**lp**

**mail**

**news**

За допомогою параметра **-d**ми визначили, що роль роздільника полів буде грати символ двокрапки.

**Злиття рядків з файлів**

Команда **paste**виконує операцію, зворотну команді **cut**. Замість вилучення колонок тексту з файлу вона додає одну або кілька колонок тексту в файл. Для цього вона читає вміст декількох файлів, комбінує поля, знайдені в них, і виводить результат в стандартний висновок. Подібно програмі **cut**, **paste**приймає кілька файлів в аргументах і/або дані зі стандартного введення.

**Об'єднання рядків з двох файлів за загальним полем**

Програма **join**діє подібно **paste**, в тому сенсі, що вона додає колонки в файл, але робить це по-своєму. Операція **join**у багатьох асоціюється з реляційними базами даних, де вона об'єднує записи з декількох таблиць за загальним ключовим полем. Програма **join**виконує ту ж операцію. Вона об'єднує дані з множини файлів, спираючись на загальне ключове поле.

**Лекція №7**

**«Простий та складний пошук файлів. Команди стиснення та архівування файлів. Багатозадачність у Linux. Структура процесів. Фоновий режим виконання завдань»**

Файлова система в Linux організована відповідно до певних угод, які переходили з одного покоління Unix-подібних систем в інше, але величезна кількість файлів може породжувати проблему, яка вирішується достатньо важко.

Існує 2 інструменти для пошуку файлів в системі:

* **locate**- виконує пошук файлів по іменах .
* **find**- виконує пошук файлів в ієрархії каталогів .

**Простий спосіб пошуку файлів**

Програма **locate**виконує швидкий пошук в базі даних імен файлів і виводить всі імена, відповідні шуканому рядку. Припустимо, наприклад, що потрібно знайти всі програми, що починаються з **zip**. Оскільки потрібно знайти програми, можна припустити, що ім'я каталогу з програмами закінчується на **bin/**. Відповідно можна спробувати виконати пошук за допомогою locate, як показано нижче:

**$ locate bin/zip**

**Locate** виконає пошук в базі даних імен файлів і виведе всі імена, що містять рядок **bin/zip**:

**/usr/bin/zip**

**/usr/bin/zipcloak**

**/usr/bin/zipgrep**

**/usr/bin/zipinfo**

**/usr/bin/zipnote**

**/usr/bin/zipsplit**

Якщо до результатів пошуку пред'являються більш суворі вимоги, команду **locate**можна об'єднати з іншими інструментами, такими як **grep**, що дозволяють здійснити більш складний пошук.

У деяких дистрибутивах при спробі запустити **locate**відразу після установки вона зазнає невдачі, але якщо спробувати використвати її на наступний день, все, як не дивно буде працювати як треба. Так в чому ж проблема? База даних для **locate**створюється іншою програмою з ім'ям **updatedb**. Зазвичай вона періодично запускається як завдання cron; тобто вона запускається системним планувальником cron через регулярні інтервали часу. У більшості систем, до складу яких входить **locate**, програма **updatedb**запускається один раз на добу. Оскільки база даних не оновлюється безперервно, швидше за все, **locate**не знаходить найсвіжіші файли. Щоб вирішити цю проблему, потрібно запустити програму **updatedb**вручну від імені суперкористувача.

**Складний спосіб пошуку файлів**

На відміну від програми **locate**, яка виконує пошук файлів по іменах, програма **find** шукає файли згідно із заданими атрибутам в зазначеному каталозі (і у вкладених підкаталогах). Команда **find**має безліч цікавих особливостей.

У найпростішому випадку програмі **find**можна передати одне або кілька імен каталогів для пошуку. Наприклад, з її допомогою можна отримати список вмісту домашнього каталогу:

**$ find ~**

Для більшості активних користувачів вона видасть довгий список. Так як список виводиться в стандартне вивндення, його можна передати по конвеєру іншим програмам. Скористаємося програмою **wc**, щоб підрахувати число файлів:

**$ find ~ | wc -l**

**47068**

Вся принадність команди **find**в тому, що її можна використовувати для пошуку файлів, які відповідають певним критеріям. Вона робить це за допомогою перевірок, операцій і параметрів.

**Перевірки**

Припустимо, ми хочемо отримати список каталогів. Для цього додамо в команду наступну перевірку:

**$ find ~ -type d | wc -l**

**1695**

Додавши перевірку **-type d**, ми обмежилися пошуком тільки каталогів. Але точно так само можна обмежити пошук тільки звичайними файлами:

**$ find ~ -type f | wc -l**

**38737**

У табл. 4.1 перераховані перевірки типів файлів, найбільш часто використовуваних з командою find.

Табл. 4.1. Перевірки типів файлів в **find**

|  |  |
| --- | --- |
| **Тип файлів** | **опис** |
| **b** | Спеціальний файл блочного пристрою |
| **c** | Спеціальний файл символьного пристрою |
| **d** | Каталог |
| **f** | звичайний файл |
| **l** | символічна посилання |

Додавши додаткові перевірки, можна виконувати пошук файлів за розміром і імені. Давайте знайдемо всі звичайні файли з іменами, відповідними шаблоном \* .JPG, і мають розмір більше 1 мегабайта:

**$ find ~ -type f -name "\* .JPG" -size + 1M | wc -l**

**840**

У цьому прикладі ми додали перевірку **-name**з шаблоном імені файлу. Шаблон укладений в лапки, щоб запобігти підстановку імен файлів командною оболонкою. Далі ми додали перевірку **-size**з рядком **+1M**. Початковий символ «плюс» вказує, що потрібно шукати файли, розмір яких перевищує вказане число. Початковий символ «мінус» змінив би значення рядка на протилежне: «менше зазначеного числа». Число без знака означає: «в точності відповідає значенню». Буква **M**в кінці визначає одиниці виміру - мегабайти (Megabytes). У табл. 4.2 перераховані символи, які можна використовувати для позначення одиниць виміру.

Табл. 4.2. Одиниці виміру, підтримувані командою **find**

|  |  |
| --- | --- |
| **символ** | **Одиниця виміру** |
| **b** | Блоки розміром по 512 байт (використовується за умовчанням, якщо інше не зазначено явно) |
| **c** | Байти |
| **w** | 2-байтні слова |
| **k** | Кілобайти (Kilobytes, блоки по 1024 байт) |
| **M** | Мегабайти (Megabytes, блоки по 1 048 576 байт) |
| **G** | Гігабайти (Gigabytes, блоки по 1 073 741 824 байт) |

Команда **find**підтримує безліч різноманітних перевірок. У табл. 7.3 наводиться короткий опис найбільш часто використовуваних з них. Зверніть увагу, що у випадках, коли потрібно числовий аргумент, допустимо використання форми записи з символами **+**і **-**, що обговорювалася вище.

Табл. 4.3. Перевірки, які підтримуються командою **find**

|  |  |
| --- | --- |
| **Перевірка** | **опис** |
| **-cmin n** | Відповідає файлам або каталогам, вміст або атрибути яких останній раз змінювалися точно **n** хвилин тому. Щоб висловити умову «менш n хвилин тому», потрібно використовувати **-n**; щоб висловити умову «більше n хвилин тому», потрібно використовувати **+ n** |
| **-cnewer ім'я** | Відповідає файлам або каталогам, вміст або атрибути яких останній раз змінювалися пізніше, ніж у файлу з вказаним ім'ям |
| **-ctime n** | Відповідає файлам або каталогам, вміст або атрибути яких останній раз змінювалися більш ніж **n \* 24**години назад |
| **-empty** | Відповідає порожнім файлам і каталогам |
| **-group група** | Відповідає файлам або каталогам, що належать зазначеній групі. Група може здаватися ім'ям або числовим ідентифікатором групи |
| **-iname шаблон** | Діє так само, як перевірка **-name**, але розрізняє регістр символів |
| **-inum n** | Відповідає файлам з номером індексного вузла (inode) n. Цю перевірку зручно використовувати для пошуку всіх жорстких посилань на певний індексний вузол |
| **-mmin n** | Відповідає файлам або каталогам, вміст яких останній раз змінювалося n хвилин тому |
| **-mtime n** | Відповідає файлам або каталогам, вміст яких останній раз змінювалося n \* 24 годин тому |
| **-name шаблон** | Відповідає файлам або каталогам, імена яких збігаються з зазначеним шаблоном |
| **-newer ім'я** | Відповідає файлам або каталогам, вміст яких останній раз змінювалося пізніше, ніж у файлу з вказаним ім'ям. Ця перевірка може стати в нагоді в сценаріях, що виконують резервне копіювання файлів. Кожен раз в процесі створення резервної копії можна оновлювати файл (наприклад, файл журналу) і потім за допомогою **find**визначати, які файли змінилися з моменту останнього оновлення |
| **-nouser** | Відповідає файлам або каталогам, що не належить якомусь допустимому користувачеві. Цю перевірку можна використовувати для пошуку файлів, що належать віддаленим облікових записів, або для виявлення слідів зловмисників |
| **-nogroup** | Відповідає файлам або каталогам, що не належить будь-якої допустимої групі |
| **-perm режим** | Відповідає файлам або каталогам з зазначеним режимом доступу. Режим може виражатися восьмеричним числом або мати символічну форму |
| **-samefile ім'я** | Діє так само, як перевірка **-inum**. Відповідає файлам з тим же номером індексного вузла (inode), що і файл з вказаним ім'ям |
| **-size n** | Відповідає файлам з розміром n |
| **-type c** | Відповідає файлів з типом c |
| **-user ім'я** | Відповідає файлам або каталогам, що належить користувачеві із зазначеним ім'ям. Аргумент ім'я може бути ім'ям або числовим ідентифікатором користувача |

**Оператори**

Незважаючи на велику кількість перевірок, підтримуваних командою **find**, ми все ще потребуємо способів визначення логічних відношень між перевірками. Наприклад, уявіть, що в деякому каталозі ми хочемо знайти всі файли і підкаталоги з небезпечними дозволами. Для цього можна було б виконати пошук всіх файлів з дозволами, що відрізняються від 0600, і каталогів з дозволами, що відрізняються від 0700. На щастя, **find**підтримує можливість комбінування перевірок за допомогою логічних операторів, з метою визначити більш складні критерії відбору. Висловити вищезгадану перевірку можна так:

**$ find ~ \ (-type f -not -perm 0600 \) -or \ (-type d -not -perm 0700 \)**

Табл. 4.4. Логічні оператори, які підтримуються командою **find**

|  |  |
| --- | --- |
| **оператор** | **опис** |
| **-and** | Відповідає, якщо виконуються умови в перевірках з обох сторін від оператора. Можна скоротити до **-a**. Зверніть увагу, що під час відсутності операторів за замовчуванням мається на увазі **-and** |
| **-or** | Відповідає, якщо виконується умова з однієї зі сторін від оператора. Можна скоротити до **-o** |
| **-not** | Відповідає, якщо умова в перевірці, наступного за оператором, не виконується. Можна скоротити до -! |
| **()** | Групують перевірки і оператори для формування великих виразів. Використовуються для управління порядком перевірок. За замовчуванням перевірки виконуються зліва направо. Часто використовуються для зміни порядку перевірок за замовчуванням, щоб отримати бажаний результат. Навіть якщо дужки не потрібні, іноді корисно включати їх, щоб зробити команди більш наочними. Не забувайте, що круглі дужки мають спеціальне значення для командної оболонки, тому їх потрібно екранувати, щоб вони передавалися команді **find**як аргументи. Зазвичай екранування виконують за допомогою символу зворотного слеша |

**Операції**

Існує безліч зумовлених операцій і кілька способів застосування операцій, визначених користувачем (табл. 4.5).

Табл. 4.5. Зумовлені операції, підтримувані командою **find**

|  |  |
| --- | --- |
| **операція** | **опис** |
| **-delete** | Видаляє поточний знайдений файл |
| **-ls** | Діє еквівалентно команді **ls –dils** щодо знайденого файлу. Результат виводиться в стандартне виведення |
| **-print** | Виводить повний шлях до знайденого файлу в стандартне виведення. Ця операція виконується за замовчуванням, якщо не вказана ніяка інша |
| **-quit** | Завершує виконання команди після виявлення першого збігу |

Підтримуваних операцій набагато більше, ніж показано тут. Повний список можна знайти на сторінці довідкового керівництва ( **man**) для команди **find**.

У нашому першому прикладі ми виконали команду:

**find ~**

Вона виводить список всіх файлів і підкаталогів, що зберігаються в домашньому каталозі. Список виводиться просто тому, що під час відсутності інших операцій передбачається операція **-print**. Тобто цю команду можна було б висловити так:

**find ~ -print**

Програму **find**можна використовувати для видалення файлів, які відповідають певним критеріям. Наприклад, наступна команда видалить всі файли з розширенням. **BAK**(яке часто використовується для позначень резервних копій файлів):

**find ~ -type f -name '\* .BAK' -delete**

Ця команда знайде в домашньому каталозі (і у вкладених підкаталогах) користувача всі файли з розширенням **.BAK**і видалить їх.

**Параметри**

Параметри допомагають керувати областю пошуку. Вони можуть включатися в вираження команди **find**поряд з іншими перевірками і операціями. У табл. 4.6 перераховані найбільш часто використовувані параметри.

Табл. 4.6. Параметри команди **find**

|  |  |
| --- | --- |
| **параметр** | **опис** |
| **-depth** | Вимагає від **find**обробити спочатку файли в каталогах і тільки потім каталоги. Цей параметр автоматично застосовується з операцією **-delete** |
| **-maxdepth число\_рівнів** | Встановлює максимальне число рівнів, на яке команда **find**може опускатися в дереві каталогів, виконуючи перевірки та операції |
| **-mindepth число\_рівнів** | Встановлює мінімальне число рівнів, на яке команда **find**повинна опуститися в дереві каталогів перед виконанням перевірок і операцій |
| **-mount** | Вимагає від **find**не виконувати обхід каталогів, в які змонтовані інші файлові системи |
| **-noleaf** | Вимагає від **find** не оптимізований пошук, спираючись на припущення, що пошук ведеться в Unix-подібній файловій системи. Цей параметр необхідно використовувати при обході файлових систем DOS / Windows CD-ROM |

**Стиснення файлів**

Протягом всієї історії розвитку обчислювальних технологій не припинялися спроби розміщення більшої кількості даних в меншому обсязі, будь то пам'ять, пристрої зберігання або смуга пропускання мережі. Багато пристроїв і технології, що міцно ввійшли в ужиток, такі як переносні плеєри, телебачення високої чіткості або широкосмуговий доступ в Інтернет, зобов'язані своїм існуванням ефективних технологій **стиснення даних** .

Алгоритми стиснення (математичні методики, що застосовуються для здійснення стиснення) діляться на дві основні категорії: **без втрат (lossless)** і **з втратами (lossy)** . Стиснення без втрат гарантує збереження всіх даних, що містяться в оригіналі. Тобто після відновлення файлу з стислій версії відновлений файл буде мати в точності той же вміст, що і нестислий оригінал. Стиснення з втратами, з іншого боку, видаляє деякі дані під час стиснення, щоб забезпечити більш високий ступінь стиснення. Відновлений файл в цьому випадку не буде збігатися з оригінальною версією, скоріше він буде близькою апроксимацією оригіналу. Прикладами стиснення з втратами можуть служити формат JPEG (для зображень) і MP3 (для музичних творів). У подальшому обговоренні ми будемо розглядати тільки стиснення без втрат, оскільки більшість даних в комп'ютерах втрат не допускає.

Програма **gzip**використовується для стиснення одного або декількох файлів. Під час роботи вона заміщає оригінальний файл його стислій версією. Відповідна програма **gunzip**використовується для відновлення стиснених файлів до вихідного стану. наприклад:

**$ ls -l / etc> foo.txt**

**$ ls -l foo. \***

**-rw-r - r-- 1 me me 15738 2012-10-14 7:15 foo.txt**

**$ gzip foo.txt**

**$ ls -l foo. \***

**-rw-r - r-- 1 me me 3230 2012-10-14 7:15 foo.txt.gz**

**$ gunzip foo.txt**

**$ ls -l foo. \***

**-rw-r - r-- 1 me me 15738 2012-10-14 7:15 foo.txt**

У цьому прикладі ми створили текстовий файл з ім'ям **foo.txt**, записавши в нього список вмісту каталогу **/etc**. Далі ми запустили програму **gzip**, яка замінила оригінальний файл стислій версією з ім'ям **foo.txt.gz**. Зі списку вмісту каталогу, який був отриманий з використанням шаблону **foo.\* ,**Можна бачити, що вихідний файл дійсно був заміщений стислою версією, і ця стисла версія вийшла майже в п'ять разів менше оригіналу. Можна також зауважити, що стиснений файл має такі ж дозволи і час, що і оригінал.

Далі ми запустили програму **gunzip**, щоб розпакувати файл. Після цього, як бачите, стисла версія була заміщена оригіналом, і знову з тими ж дозволами і часом.

Програма **gzip**має безліч параметрів, частина яких описана в табл. 4.7.

Табл. 4.7. Параметри команди **gzip**

|  |  |
| --- | --- |
| **параметр** | **опис** |
| **-c** | Виводить результат на стандартне виведення і зберігає оригінальні файли. Аналогічний ефект дають параметри **--stdout**і **--to-stdout** |
| **-d** | Розпаковує файл. З цим параметром **gzip**діє як **gunzip**. Аналогічний ефект дають параметри **--decompress**і **--uncompress** |
| **-f** | Примусове (force) стиснення, навіть якщо стисла версія оригінального файлу вже існує. Аналогічний ефект дає параметр **--force** |
| **-h** | Виводить інформацію про порядок використання. Аналогічний ефект дає параметр **--help** |
| **-l** | Виводить список статистик для кожного стисненого файлу. Аналогічний ефект дає параметр **--list** |
| **-t** | Перевіряє цілісність стисненого файлу. Аналогічний ефект дає параметр **--test** |
| **-v** | Виводить в процесі роботи повідомлення з інформацією про хід стиснення. Аналогічний ефект дає параметр **--verbose** |
| **число** | Встановлює ступінь стиснення. Числом може бути будь-який цілочисельне значення в діапазоні від 1 (висока швидкість роботи, низький ступінь стиснення) до 9 (низька швидкість роботи, високий ступінь стиснення). Значення 1 і 9 можна також замінити параметрами **--fast**і **--best**відповідно. За замовчуванням використовується значення 6 |

Разом з **gzip**зазвичай поставляється програма **zcat**. Вона застосовується до файлів, стисненим за допомогою **gzip**, за аналогією з командою **cat**:

**$ zcat foo.txt.gz | less**

Існує також програма **zless**. Вона замінює собою конвеєр, представлений вище.

**Високий ступінь стиснення ціною швидкості**

Програма **bzip2**Джуліана Сюарда схожа на програму **gzip**, але використовує інший алгоритм, який забезпечує більш високу ступінь стиснення ціною зниження швидкості роботи. Багато в чому вона діє так само, як **gzip**. Файл, стиснений за допомогою **bzip2**, отримує розширення **.bz2**:

**$ ls -l / etc> foo.txt**

**$ ls -l foo.txt**

**-rw-r - r-- 1 me me 15738 2012-10-17 13:51 foo.txt**

**$ bzip2 foo.txt**

**$ ls -l foo.txt.bz2**

**-rw-r - r-- 1 me me 2792 2012-10-17 13:51 foo.txt.bz2**

**$ bunzip2 foo.txt.bz2**

Всі параметри програми **gzip**(крім **-r**), представлені вище, підтримуються також програмою **bzip2**. Але потрібно мати на увазі, що параметр ступеня стиснення (**число**) має дещо інший сенс для **bzip2**. У парі з **bzip2**поставляються рограми **bunzip2**і **bzcat**для розпаковування файлів. Існує також програма **bzip2recover**для відновлення пошкоджених файлів формату **.bz2**.

**Архівування файлів**

Часто разом із стисненням використовується операція архівування. Архівування - це процес збору множини файлів і пакування їх в один великий файл. Архівування часто застосовується як один з етапів створення резервних копій системи. Воно також використовується при переміщенні старих даних з системи в деяке довгострокове сховище.

У світі програмного забезпечення для Unix-подібних систем існує програма **tar**- класичний інструмент для архівації файлів. Її ім'я, яке розшифровується як **tape archive** (архів на магнітній стрічці), вказує, що спочатку інструмент призначався для створення архівів на магнітних стрічках. Він до цих пір використовується для вирішення цієї традиційної задачі, але з неменшим успіхом підтримує інші пристрої зберігання. Нам часто доводиться бачити імена файлів з розширенням **.tar**або **.tgz**, які позначають «прості» tar-архіви і архіви, стислі за допомогою **gzip**відповідно. Архів може складатися з групи окремих файлів, ієрархій каталогів або і того і іншого. Команда **tar**повинен виглядати так:

**tar режим [параметри] шлях ...**

де під режимом мається на увазі один з декількох режимів роботи, перерахованих в табл. 4.8 (тут представлені не всі параметри; повний список можна знайти на сторінці довідкового керівництва ( **man**) для **tar**).

Табл. 4.8. Режими команди **tar**

|  |  |
| --- | --- |
| **режим** | **опис** |
| **-c** | Створити архів зі списку файлів і / або каталогів |
| **-x** | Витягти файли з архіву |
| -r | Додати вказаний файл і / або каталог у кінець архіву |
| **-t** | Вивести список вмісту архіву |

Створимо архів всієї папки **playground**:

**$ tar -cf playground.tar playground**

Режим і параметр **-f**, який використовується для визначення імені tar-архіву, можна об'єднувати, і при цьому не потрібно використовувати початковий дефіс. Але потрібно мати на увазі, що режим завжди повинен вказуватися першим, перед будь-якими параметрами. Подивитися вміст архіву можна за допомогою наступної команди:

**$ tar -tf playground.tar**

Для отримання більш докладного списку додамо параметр **-v**(verbose - подробиці):

**$ tar -tvf playground.tar**

**Пакування і стиснення файлів**

Програма **zip**одночасно є і інструментом стиснення, і архиватором. Формат файлів, який використовується програмою, знайомий користувачам Windows - програма читає і створює файли з розширенням **.zip**. Однак в Linux частіше інших використовується програма стиснення **gzip**, а на другому місці **bzip2**. Користувачі Linux використовують **zip** в основному для обміну файлами з системами Windows, а не як основний інструмент стиснення та архівування.

У найпростішому випадку програма zip має наступний синтаксис:

**zip параметри стиснений\_файл файл ...**

Наприклад, нижче показано, як створити zip-архів папки **playground**:

**$ zip -r playground.zip playground**

Без параметра -r (відповідає за рекурсивний обхід каталогів) в архів буде включенийтільки каталог playground (без свого вмісту). Розширення **.zip**додається до імені вихідного файлу автоматично, а ми включили його в приклад для наочності.

У процесі створення zip-архіву програма **zip**зазвичай виводить послідовність повідомлень, як показано нижче:

**adding: playground / dir-020 / file-Z (stored 0%)**

**adding: playground / dir-020 / file-Y (stored 0%)**

**adding: playground / dir-020 / file-X (stored 0%)**

**adding: playground / dir-087 / (stored 0%)**

**adding: playground / dir-087 / file-S (stored 0%)**

Ці повідомлення показують стан кожного файлу, доданого в архів. **zip**додає файли в архів, використовуючи один з двох методів: або «store» (просте збереження) - без стиснення, як в прикладі, наведеному вище, або «deflate» - зі стисненням. Числове значення, що випливає за назвою методу додавання, вказує досягнуту ступінь стиснення. Витяг вмісту з zip-архіву виконується просто - за допомогою програми **unzip.**

**Процеси**

Сучасні операційні системи зазвичай є багатозадачними, в тому сенсі, що створюють ілюзію одночасного вирішення множини завдань, швидко переключаючись з виконання однієї програми на іншу. Ядро Linux керує всім цим за допомогою процесів. Саме за допомогою процесів Linux організовує припинення програм в очікуванні, поки настане їхня черга використовувати процесор.

У момент запуску системи ядро ініціює виконання кількох власних завдань у вигляді процесів і запускає програму з назвою **init** . У свою чергу **init** виконує послідовність сценаріїв командної оболонки (знаходяться в **/etc**), званих **сценаріями початкового завантаження (init scripts),** які запускають всі системні служби. Багато з цих служб реалізовані як **програми-демони (daemon programs)**, тобто програми, що діють у фоновому режимі і виконують свою роботу без участі користувача. Тому, навіть за відсутності зареєстрованих користувачів система виконує певні службові процедури.

Принцип, за яким програма може запускати інші програми, виражається правилом: батьківський процес запускає дочірній процес.

Ядро зберігає інформацію про всі процеси, щоб упорядкувати їх роботу. Наприклад, кожному процесу присвоюється номер, який називають **ідентифікатором процесу (Process ID, PID)** . Ідентифікатори процесів присвоюються в порядку зростання, при цьому процес **init** завжди отримує ідентифікатор PID **1** . Ядро також стежить за пам'яттю, виділеної кожному процесу, і за готовністю процесів відновити виконання. Подібно файлів, процеси також мають ідентифікатори власника і користувача, ефективний (або діючий) ідентифікатор користувача і т. д.

**Перегляд списку процесів**

Найчастіше для перегляду списку процесів використовується команда **ps**. Програма **ps**має безліч параметрів, але в самому простому випадку вона використовується наступним чином:

**$ ps**

**PID TTY TIME CMD**

**5198 pts / 1 00:00:00 bash**

**10129 pts / 1 00:00:00 ps**

У цьому прикладі команда вивела список з двома процесами: процес 5198 і процес 10129 - програми **bash**і **ps**відповідно. Як можна помітити, за замовчуванням **ps**виводить не дуже багато інформації, тільки процеси, пов'язані з поточним сеансом. Щоб побачити більше, слід передати додаткові параметри, але перш ніж ми зробимо це, давайте розглянемо інші поля у висновку команди **ps**. Поле **TTY**- це скорочення від **teletype (телетайп),** воно містить інформацію про керуючий термінал процесу. В Unix в цьому полі виводиться тип терміналу. Поле **TIME**містить обсяг процесорного часу, спожитого процесом. Як бачите, жоден з процесів не є надто обтяжливим для комп'ютера.

Якщо додати параметр **x**, можна отримати більш багату інформацію про те, що відбувається в системі:

**$ ps x**

**PID TTY STAT TIME COMMAND**

**2799? Ssl 0:00 / usr / libexec / bonobo-activation-server -ac**

**122 Глава 10. Процеси**

**2820? Sl 0:01 /usr/libexec/evolution-data-server-1.10 -**

**15647? Ss 0:00 / bin / sh / usr / bin / startkde**

**15751? Ss 0:00 / usr / bin / ssh-agent / usr / bin / dbus-launch -**

**15754? S 0:00 / usr / bin / dbus-launch --exit-with-session**

**15755? Ss 0:01 / bin / dbus-daemon --fork --print-pid 4 -pr**

**15774? Ss 0:02 / usr / bin / gpg-agent -s -daemon**

**15793? S 0:00 start\_kdeinit --new-startup + kcminit\_start**

**15794? Ss 0:00 kdeinit Running ...**

**15797? S 0:00 dcopserver -nosid**

Додатковий параметр **x**(зверніть увагу на відсутність дефіса) повідомляє команді **ps**, що та повинна вивести всі процеси, незалежно від того, які термінали (якщо такі є) керують ними. Символ **?**в поле **TTY**вказує на відсутність керуючого терміналу. Таким чином, параметр x дозволяє побачити всі процеси в системі, якими ми володіємо.

Так як в системі одночасно виконується безліч процесів, **ps**виробляє досить довгі списки. Часто буває корисно передати висновок **ps**команді **less**через конвеєр, щоб його простіше було переглядати. Деякі комбінації параметрів приводять до виведення дуже довгих рядків, тому незайвим буде також відчинити вікно емулятора термінала на весь екран.

У цьому прикладі у висновку з'явився новий стовпець - **STAT**. Назва **STAT**- це скорочення від **state (стан),** стовпець містить інформацію про поточний стан процесу, як показано в табл. 4.9.

Табл. 4.9. Cтани процесів

|  |  |
| --- | --- |
| **стан** | **значення** |
| **R** | Виконується. Процес виконується або готовий до виконання |
| **S** | Призупинений. Процес тимчасово не виконується; швидше за все, знаходиться в очікуванні певної події, такого як натискання клавіші або прибуття мережевого пакету |
| **D** | Призупинений без можливості переривання. Процес очікує завершення операції введення/виведення, наприклад, дисковим пристроєм |
| **T** | Зупинений. Процес примусово зупинено (докладніше про це розповідається нижче) |
| **Z** | Недіючий процес-«зомбі». Це дочірній процес, який завершився, але не був видалений батьком |
| **<** | Високопріоритетний процес. Існує можливість найбільш важливим процесам виділити більше процесорного часу. Дана властивість процесу називається **niceness** (поступливість). Про процеси з більш високими пріоритетами кажуть, що вони менш поступливі, тому що споживають більше процесорного часу, залишаючи менше іншим процесам |
| **N** | Фоновий процес. Процес з низьким пріоритетом (або поступливий процес) отримує процесорний час тільки після того, як будуть обслужені процеси з більш високим пріоритетом |

Символ, що описує стан процесу, може супроводжуватися іншими символами. Вони відображають деякі екзотичні характеристики процесів. За додатковою інформацією звертайтеся до сторінці довідкового керівництва (**man**) для **ps**.

Ще одна популярна комбінація параметрів - **aux**(без дефіса на початку). Вона дозволяє отримати ще більше інформації:

**$ ps aux**

**USER PID% CPU% MEM VSZ RSS TTY STAT START TIME COMMAND**

**root 1 0.0 0.0 2136 644? Ss Mar05 0:31 init**

**root 2 0.0 0.0 0 0? S <Mar05 0:00 [kt]**

**root 3 0.0 0.0 0 0? S <Mar05 0:00 [mi]**

**root 4 0.0 0.0 0 0? S <Mar05 0:00 [ks]**

**root 5 0.0 0.0 0 0? S <Mar05 0:06 [wa]**

**root 6 0.0 0.0 0 0? S <Mar05 0:36 [ev]**

**root 7 0.0 0.0 0 0? S <Mar05 0:00 [kh]**

**Перегляд стану процесів в динаміці**

Команда **ps**надає масу інформації про те, що робиться в комп'ютері, але вона дає тільки миттєвий знімок, тобто інформація, що повертається нею, є дійсною лише на момент виклику команди. Щоб побачити роботу комп'ютера в динаміці, скористаємося командою **top**:

**$ top**

Програма **top**постійно оновлює інформацію про процеси (за замовчуванням з періодом, рівним 3 секундам), щоб показати їх активність з плином часу. Ім'я програми **top**відображає той факт, що вона використовується для перегляду «топа» (найбільш активних) процесів в системі. Виведення команди **top** ділиться на дві частини: зведена інформація про систему і таблиця процесів, відсортованих по споживанню ними процесора:

**top - 14:59:20 up 6:30, 2 users, load average: 0.07, 0.02, 0.00**

**Tasks: 109 total, 1 running, 106 sleeping, 0 stopped, 2 zombie**

**Cpu (s): 0.7% us, 1.0% sy, 0.0% ni, 98.3% id, 0.0% wa, 0.0% hi, 0.0% si**

**Mem: 319496k total, 314860k used, 4636k free, 19392k buff**

**Swap: 875500k total, 149128k used, 726372k free, 114676k cach**

**PID USER PR NI VIRT RES SHR S% CPU% MEM TIME + COMMAND**

**6244 me 39 19 31752 3124 2188 S 6.3 1.0 16: 24.42 trackerd**

**11071 me 20 0 2304 1092 840 R 1.3 0.3 0: 00.14 top**

**6180 me 20 0 2700 1100 772 S 0.7 0.3 0: 03.66 dbus-dae**

**6321 me 20 0 20944 7248 6560 S 0.7 2.3 2: 51.38 multiloa**

**4955 root 20 0 104m 9668 5776 S 0.3 3.0 2: 19.39 Xorg**

**1 root 20 0 2976 528 476 S 0.0 0.2 0: 03.14 init**

**2 root 15 -5 0 0 0 S 0.0 0.0 0: 00.00 kthreadd**

**3 root RT -5 0 0 0 S 0.0 0.0 0: 00.00 migratio**

**4 root 15 -5 0 0 0 S 0.0 0.0 0: 00.72 ksoftirq**

**5 root RT -5 0 0 0 S 0.0 0.0 0: 00.04 watchdog**

**6 root 15 -5 0 0 0 S 0.0 0.0 0: 00.42 events / 0**

**7 root 15 -5 0 0 0 S 0.0 0.0 0: 00.06 khelper**

**41 root 15 -5 0 0 0 S 0.0 0.0 0: 01.08 kblockd /**

**67 root 15 -5 0 0 0 S 0.0 0.0 0: 00.00 kseriod**

**114 root 20 0 0 0 0 S 0.0 0.0 0: 01.62 pdflush**

**116 root 15 -5 0 0 0 S 0.0 0.0 0: 02.44 kswapd0**

**Управління процесами**

**Переривання процесу**

Комбінація **CTRL+C**в терміналі перериває виконання програми. Таким способом можна перервати виконання багатьох (але не всіх) програм командного рядка.

**Переведення процесів в фоновий режим**

Щоб запустити програму відразу в фоновому режимі, потрібно додати в кінець команди символ амперсанда (&):

**$ Xlogo &**

**[1] 28236**

**$**

Після введення такої команди на екрані з'явиться вікно **xlogo**, а командна оболонка повернеться в запрошення до введення, але перед цим виведе таємничі числа. Це повідомлення є частиною **механізму управління завданнями (job control)** . Таким способом командна оболонка повідомляє, що ми запустили завдання з номером 1 ([1]) і воно отримало ідентифікатор процесу PID 28236. Якщо тепер виконати команду **ps**, можна побачити цей процес:

**$ ps**

**PID TTY TIME CMD**

**10603 pts / 1 00:00:00 bash**

**28236 pts / 1 00:00:00 xlogo**

**28239 pts / 1 00:00:00 ps**

Механізм управління завданнями також дає можливість вивести список завдань, запущених в терміналі. Цей список можна отримати командою **jobs**:

**$ jobs**

**[1] + Running xlogo &**

Результати показують, що у нас є одне що виконується завдання з номером 1, яке було запущено командою **xlogo &**.

**Повернення процесу на передній план**

Процес у фоновому режимі не отримує введення з клавіатури, в тому числі не бачить спроб перервати його комбінацією **CTRL+C**. Повернути процес на передній план можна командою **fg**, як в наступному прикладі:

**$ jobs**

**[1] + Running xlogo &**

**$ fg %1**

**xlogo**

За командою **fg**повинен слідувати знак відсотка і номер завдання (ця комбінація називається специфікатором завдання, або jobspec). Якщо є тільки одне фонове завдання, специфікатор можна опустити. Тепер завершимо xlogo введенням **CTRL+C**.

**Призупинення процесу**

Іноді необхідно призупинити процес на час, але не завершити його. Це часто робиться з метою перевести процес переднього плану в фоновий режим. Щоб призупинити процес переднього плану, використовуйте комбінацію **CTRL+Z**:

**$ xlogo**

**[1] + Stopped xlogo**

**$**

Далі можна або повернути програму на передній план командою **fg**, або перевести її в фоновий режим командою **bg**:

**$ bg %1**

**[1] + xlogo &**

**$**

Так само як у випадку з командою **fg**, специфікатор завдання можна опустити, якщо є тільки одне завдання.

Можливість переведення у фоновий режим корисна і в тому випадку, якщо при запуску програми з графічним інтерфейсом з командного рядка ви забули додати в кінець команди символ **&.**

Навіщо може знадобитися запускати програму з графічним інтерфейсом з командного рядка? Для цього є дві причини. По-перше, необхідна програма може бути відсутнім в меню програм оточення робочого столу (як, наприклад, **xlogo**).

По-друге, запускаючи програму з командного рядка, можна побачити повідомлення про помилки, які невидимі, коли програма запускається з графічного інтерфейсу. Іноді програма аварійно завершується при запуску з графічного меню. В цьому випадку, запускаючи її з командного рядка, можна за повідомленнями про помилки зрозуміти причину аварії. Крім того, деякі програми з графічним інтерфейсом мають цікаві параметри командного рядка.

**Зупинка процесів**

Однією з вкрай важливих завдань системного адміністратора є виконання своєчасної та правильної зупинки процесів. Іноді процес призупиняється і вимагає лише невеликого втручання для продовження своєї роботи або повного зупинення. Бувають і такі ситуації, що процес стає некерованим, безмірно поглинає процесорний час і не може бути припинений звичайними способами. І в тому і в іншому випадку для управління процесом необхідно скористатися певною командою. В системі Linux використовується такий же спосіб організації взаємодії між процесами, як і в системі UNIX.

У Linux процеси обмінюються один з одним даними з використанням сигналів. Сигнал процесу - це стандартне повідомлення, що розпізнається процесом, після отримання якого процес може вибрати, ігнорувати його чи виконати відповідну йому дію. Застосовувані в процесі способи обробки сигналів закладаються розробниками при написанні програми. У більшості якісно написаних додатків передбачена здатність отримувати стандартні сигнали процесу UNIX і діяти відповідно до цих сигналів. Зазначені сигнали наведені в табл. 7.10.

Табл. 7.10. Сигнали процесу Linux

|  |  |  |
| --- | --- | --- |
| **сигнал** | **ім'я** | **опис** |
| **1** | **HUP** | Відбій |
| **2** | **INT** | Переривання |
| **3** | **QUIT** | Припинити виконання |
| **9** | **KILL** | Беззастережно завершити |
| **11** | **SEGV** | Порушення сегмента |
| **15** | **TERM** | Завершити, якщо можливо |
| **17** | **STOP** | Беззастережно зупинити, але не завершувати |
| **18** | **TSTP** | Зупинити або призупинити, але продовжити роботу у фоновому режимі |
| **19** | **CONT** | Відновити виконання після сигналів STOP або TSTP |

В системі Linux передбачені дві описані нижче команди, які дозволяють відправляти сигнали працюючим процесам.

Команда **kill** дає можливість відправляти сигнали процесам з урахуванням їх **ідентифікатора процесу (process ID - PID).** За замовчуванням команда**kill**відправляє сигнал **TERM**всім процесам з ідентифікаторами процесів, перерахованими в командному рядку. На жаль, передбачена можливість використовувати лише **PID** процесу замість імені команди, за допомогою якої він був запущений, тому іноді завдання правильного виконання команди **kill**стає скрутною.

Щоб мати право передати сигнал процесу, необхідно або бути власником процесу, або зареєструватися як користувач **root** .

**$ kill 3940**

**-bash: kill: (3940) - Operation not permitted**

Сигнал **TERM**вказує процесу, що має бути виконана процедура коректного припинення виконання. На жаль, якщо при цьому доводиться мати справу з процесом, що вийшов з-під контролю, то цей запит, швидше за все, буде проігноровано. Якщо буде потрібно завершити виконання процесу примусово, можна скористатися параметром **-s**, що дозволяє вказувати інші сигнали (з використанням імені або номера сигналу).

Загальноприйнята процедура полягає в тому, що спочатку слід спробувати застосувати сигнал **TERM**. Якщо цей сигнал буде проігнорований процесом, то робиться спроба відправити сигнал **INT**або **HUP**. Якщо програма розпізнає такий сигнал, то спробує коректно завершити виконуються в ній операції, перш ніж зробити громовідвід. Сигналом, що визначає найвищу ступінь примусового завершення, є сигнал **KILL**. Після отримання цього сигналу процес негайно припиняє виконання. Це може призвести до спотворення інформації в файлах.

Як показує наступний приклад, з командою **kill**не пов'язаний будь-який висновок:

**# kill -s HUP 3940**

Для визначення того, виявилася чи результативною ця команда, необхідно ще раз викликати на виконання команду **ps**або **top**, щоб дізнатися, чи вдалося зупинити процес, що порушує нормальну роботу.

Команда **killall**являє собою потужний спосіб зупинки процесів з використанням їхніх імен, а не числових значень **PID** . Команда **killall**дозволяє також використовувати символи-шаблони, тому стає вельми корисним інструментом при відновленні нормальної роботи системи, в якій виникли неприпустимі процеси:

**# killall http \***

Цей приклад демонструє знищення всіх процесів, імена яких починаються з http, зокрема служб httpd для веб-сервера Apache.

**Планування регулярного виконання завдань**

В системі Linux передбачена програма **cron**, яка дозволяє планувати завдання, призначені для виконання на регулярній основі. Програма **cron** виконується в фоновому режимі і перевіряє спеціальні таблиці, звані таблицями **cron** , на наявність завдань, які намічені для виконання.

У таблиці **cron** використовується спеціальний формат, що дозволяє вказувати, коли має бути виконано завдання. Формат таблиці **cron** є наступним:

**min hour dayofmonth month dayofweek command**

Таблиця **cron**дозволяє задавати в ній записи в якості конкретних значень і діапазонів значень (таких як 1-5), а також символів-шаблонів (зірочок). Наприклад, якщо деяка команда повинна виконуватися щодня о 10:15, то можна скористатися наступною записом в таблиці cron:

**15 10 \* \* \* command**

Символи-шаблони, які використовуються в полях **dayofmonth**, **month** і **dayofweek**, вказують, що програма **cron** повинна виконувати команду в кожен день кожного місяця о 10:15. А для вказівки, наприклад, що команда повинна виконуватися о 4:15 пополудні в щопонеділка, можна використовувати такий запис:

**15 16 \* \* 1 command**

Елемент **dayofweek** може бути заданий у вигляді трехсімвольного текстового значення (mon, tue, wed, thu, fri, sat, sun) або числового значення з ряду значень, такого, що 0 відповідає неділі, а 6 - суботи.

Нижче наведено ще один приклад, який забезпечує виконання команди о 12:00 після полудня в першу добу кожного місяця, для чого використовується наступний формат:

**00 12 1 \* \* command**

Елемент **dayofmonth** вказує значення числа місяця (1-31).

У списку команд повинні бути вказані повні імена шляхів до командам або до сценаріїв командного інтерпретатора, що підлягають виконанню. Передбачена можливість задавати будь-які параметри командного рядка або символи перенаправлення, як і в звичайній командному рядку:

**15 10 \* \* \* / home / rich / test4> test4out**

Програма **cron** виконує сценарій із застосуванням облікового запису користувача, який передав завдання на виконання. Таким чином, користувач повинен мати належні дозволи на доступ до команди і до вихідних файлів, зазначеним у виклику команди.

**Побудова таблиці cron**

Кожен користувач системи (включаючи користувача **root** ) може мати власну таблицю **cron**, призначену для виконання запланованих завдань. У Linux для обробки таблиці **cron** передбачена команда **crontab**. Для виведення існуючої таблиці **cron** на зовнішній пристрій використовується параметр **-l**:

**$ crontab -l**

**no crontab for rich**

За замовчуванням файл таблиці **cron** не створюється автоматично для кожного користувача. Для додавання записів в таблицю **cron** застосовується параметр **-e**. Команда **crontab**після виклику з цим параметром запускає текстовий редактор стосовно існуючої таблиці **cron** (або до порожнього файлу, якщо таблиця **cron** ще не була створена).

**Лекція №5**

**«Написання сценаріїв bash, їх створення та виклик. Робота зі змінними у сценарії bash. Код завершення виконання сценарію»**

У попередніх лекціях було показано, як працювати з інтерфейсом командного рядка - запрошенням командного інтерпретатора, вводячи окремі команди і переглядаючи результати команд. У порівнянні з цим основною особливістю сценаріїв командного інтерпретатора є можливість вводити кілька команд і послідовно обробляти результати, сформовані кожною командою, причому надається навіть можливість передавати результати від однієї команди до іншої. Командний інтерпретатор дозволяє складати ланцюжки команд так, що їх виконання відбувається за один крок.

Якщо потрібно, щоб дві команди були виконані разом, їх можна ввести в одному й тому ж рядку запрошення командного інтерпретатора, розділивши крапкою з комою:

**$ date; who**

**Mon Feb 21 15:36:09 EST 2011**

**Christine tty2 2011-02-21 15:26**

**Samantha tty3 2011-02-21 15:26**

**Timothy tty1 2011-02-21 15:26**

**user tty7 2011-02-19 14:03 (0)**

**user pts / 0 2011-02-21 15:21 (: 0.0)**

Ми тільки що написали сценарій командного інтерпретатора. У цьому простому сценарії використовуються тільки дві команди командного інтерпретатора **bash**. Спочатку виконується команда **date**, яка показує поточні значення дати і часу, а за нею йде виведення команди **who**, яке дозволяє дізнатися, які користувачі в даний час зареєстровані в системі. Використовуючи цей спосіб, можна з'єднувати в ланцюжок будь-яку необхідну кількість команд, аж до граничного значення кількості знаків в командному рядку, рівного 255 символів.

Безумовно, застосування даного способу є досить зручним при створенні невеликих сценаріїв, але він має серйозний недолік, який полягає в тому, що доводиться знову і знову набирати весь текст ланцюжка команд в командному рядку, коли виникає необхідність викликати її на виконання. Тому була передбачена можливість об'єднувати команди в простий текстовий файл, щоб не доводилося щоразу вводити команди вручну в командному рядку. І якщо виникла необхідність виконати ці команди, досить викликати на виконання текстовий файл.

**Створення файлу сценарію**

Щоб помістити команди командного інтерпретатора в текстовий файл, спочатку необхідно скористатися текстовим редактором для створення файлу, а потім ввести команди в цей файл.

При створенні файлу сценарію командного інтерпретатора необхідно в першому рядку файлу вказати використовуваний командний інтерпретатор. Нижче наведено формат застосовується для цього рядка.

**#!/bin/bash**

У всіх інших рядках сценарію командного інтерпретатора знак дієз (#), званий також знаком фунта, розглядається як ознака початку рядка коментаря. Рядок коментаря в сценарії командного інтерпретатора не обробляється командним інтерпретатором. Але перший рядок файлу сценарію командного інтерпретатора трактується особливим чином, і знак дієз, за ​​яким слід знак оклику, повідомляє командному інтерпретатору, який командний інтерпретатор повинен бути викликаний для виконання сценарію.

Слідом за рядком із зазначенням командного інтерпретатора необхідно приступити до прогресивного введення команд в файл, вставляючи після кожного рядка символ повернення каретки. Як вже було сказано, передбачена також можливість вводити коментарі, позначаючи їх за допомогою знака решітки. Розглянемо наступний приклад:

**#!/bin/bash**

**# Цей сценарій відображає дату і показує, хто зареєстрований в системі**

**date**

**who**

У ньому знаходяться всі елементи сценарію, про які було сказано вище. У сценаріях командного інтерпретатора за бажанням також можна використовувати крапку з комою і поміщати кілька команд в одному й тому ж рядку, але зазвичай прийнято відводити для кожної команди окремий рядок. Командний інтерпретатор обробляє команди в тому порядку, в якому вони наведені в файлі.

Збережемо цей сценарій у файлі **test1**, після чого ми будемо майже готові до виконання наступних кроків. Справа в тому, що потрібно ще дещо зробити, перш ніж з'явиться можливість викликати на виконання цей новостворений файл сценарію командного інтерпретатора. Якщо ж спроба виконати цей файл буде зроблена негайно, то отримані результати нас розчарують:

**$ test1**

**bash: test1: command not found**

Перша перешкода, з якою доводиться стикатися, полягає в тому, що командному інтерпретатору **bash** потрібно якось вказати, де знаходиться файл сценарію. У командному інтерпретаторі для пошуку команд використовується змінна середовища **PATH**. Достатньо ознайомитися із значенням змінної середовища **PATH**, як відразу стає зрозумілою причина виниклої проблеми:

**$ echo $ PATH**

**/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin**

Змінна середовища **PATH**задана так, що пошук команд здійснюється тільки в декількох заданих каталогах. Щоб забезпечити для командного інтерпретатора можливість знайти сценарій test1, необхідно скористатися одним з наступних двох способів.

* Додати позначення каталогу, в якому знаходиться наш файл сценарію командного інтерпретатора, в змінну середовища **PATH**.
* Задати абсолютний або відносний шлях до файлу в якості посилання на створений нами файл сценарію командного інтерпретатора в запрошенні до введення інформації.

В даному прикладі скористаємося другим способом і передамо командному інтерпретатору точну вказівку ​​на місцезнаходження файлу сценарію. Нагадаємо, що для формування посилання на файл в поточному каталозі можна використовувати оператор одинарної точки в командному інтерпретаторі:

**$ ./test1**

**bash: ./test1: Permission denied**

Після цього командний інтерпретатор успішно знайде файл сценарію командного інтерпретатора, проте виявиться чергова проблема. Командний інтерпретатор вказує, що ви не маєте дозвіл на виконання файлу. Досить коротко ознайомитися з відомостями про права доступу до файлу, щоб зрозуміти, що відбувається:

**$ ls -l test1**

**-rw-r - r-- 1 user user 73 Sep 24 19:56 test1**

Наступний крок полягає в тому, щоб надати власнику файлу дозвіл виконувати файл з використанням команди **chmod**:

**$ chmod u+x test1**

**$ ./test1**

**Mon Feb 21 15:38:19 EST 2011**

**Christine tty2 2011-02-21 15:26**

**Samantha tty3 2011-02-21 15:26**

**Timothy tty1 2011-02-21 15:26**

**user tty7 2011-02-19 14:03 (0)**

**user pts / 0 2011-02-21 15:21 (: 0.0)**

**Відображення повідомлень**

Команди командного інтерпретатора найчастіше виробляють власний висновок, який відображається на консолі при виконанні сценарію. Проте нерідко виникає необхідність в додаванні своїх власних текстових повідомлень, які могли б допомогти користувачеві сценарію стежити за тим, які дії здійснюються в ході виконання сценарію. Для цього передбачена команда **echo**. Команда **echo**може відобразити просту текстову рядок, яка введена слідом за цією командою:

**$ echo This is a test**

**This is a test**

Слід враховувати, що за замовчуванням не потрібно використовувати лапки для позначення початку і кінця відображається рядки. Але іноді вихідні дані можуть зазнати спотворення, якщо в рядку використовуються лапки:

**$ echo Let's see if this'll work**

**Lets see if thisll work**

У команді **echo**допускається застосовувати для позначення текстових рядків одинарні або подвійні лапки. Якщо ж лапки повинні бути задані і в тексті рядки, то лапки одного типу повинні використовуватися в тексті, а лапки іншого типу - служити для позначення початку і кінця рядка:

**$ echo "This is a test to see if you're paying attention"**

**This is a test to see if you're paying attention**

**$ echo 'Rich says "scripting is easy".'**

**Rich says "scripting is easy".**

Після цього всі лапки відображаються в вихідних даних належним чином.

Допускається можливість додавати інструкції **echo**в будь-яких місцях сценаріїв командного інтерпретатора, де виникає необхідність вивести додаткові відомості:

**$ Cat test1**

**#!/bin/bash**

**# Цей сценарій відображає дату і показує, хто зареєстрований**

**# в системі**

**echo The time and date are:**

**date**

**echo "Let's see who's logged into the system:"**

**who**

**Використання змінних**

Сценарії командного інтерпретатора, які складаються з окремих команд, також важливі, але іноді виникають ситуації, коли цього стає недостатньо. Наприклад, часто виявляється необхідність включити інші дані в команди командного інтерпретатора з метою обробки інформації. Це завдання можна вирішити з використанням змінних. Змінні дозволяють зберігати на час інформацію в сценарії командного інтерпретатора для подальшого застосування в інших командах в сценарії.

**Змінні середовища**

Командний інтерпретатор підтримує змінні середовища, які відстежують конкретну системну інформацію, таку як ім'я системи, ім'я користувача, зареєстрованого в системі, ідентифікатор користувача в системі (званий просто ідентифікатором користувача - UID), застосовуваний за замовчуванням вихідний каталог користувача і шлях пошуку файлів, використовуваний командним інтерпретатором для пошуку програм. Щоб ознайомитися з повним списком активних змінних середовища, до яких може бути отриманий доступ, досить скористатися командою **set**.

Значення цих змінних середовища можна вставляти в сценарії, для чого необхідно вказати ім'я змінної середовища, якому передує знак долара. Такий спосіб доступу до змінних середовища демонструється в наступному сценарії:

**$ cat test2**

**#!/bin/bash**

**# Відображення інформації про користувача.**

**echo "User info for userid: $USER"**

**echo UID: $UID**

**echo HOME: $HOME**

**Змінні**

У сценаріях командного інтерпретатора можна не тільки використовувати змінні середовища, але також задавати і включати свої власні змінні. Завдання змінних дозволяє на час зберігати дані і використовувати їх у всьому сценарії, в результаті чого сценарій командного інтерпретатора в більшій мірі стає схожим на справжню комп'ютерну програму.

Імена для користувача змінних можуть бути будь-яким текстовим рядком довжиною до 20 символів, що складається з букв, цифр і символів підкреслення. В іменах призначених для користувача змінних чутливі до регістру, тому змінна **Var1**розглядається як відмінна від змінної **var1**. Забуваючи про це просте правило, початківці програмісти на мовах сценаріїв часто допускають помилки, важко діагностуються.

Значення присвоюються призначеним для користувача змінним за допомогою знака рівності (=). Між ім'ям змінної, знаком рівності і значенням не повинно бути жодного пропуску (недосвідчені програмісти іноді забувають і про це правило). Нижче наведено кілька прикладів присвоювання значень призначеним для користувача змінним.

**var1 = 10**

**var2 = -57**

**var3 = testing**

**var4 = "still more testing"**

При обробці сценаріїв командного інтерпретатора тип даних, використовуваний для представлення значення змінної, визначається автоматично. Змінні, які визначені в сценарії командного інтерпретатора, зберігають свої значення протягом усього часу виконання сценарію і знищуються після завершення сценарію.

Для посилання на змінні, як і на змінні середовища, застосовується знак долара:

**$ cat test3**

**#!/bin/bash**

**# Перевірка змінних**

**days=10**

**guest="Katie"**

**echo "$guest checked in $ days days ago"**

**days=5**

**guest="Jessica"**

**echo "$guest checked in $ days days ago"**

Виконання цього сценарію призводить до отримання наступного виведення:

**$ chmod u+x test3**

**$ ./test3**

**Katie checked in 10 days ago**

**Jessica checked in 5 days ago**

При кожному посиланні на змінну відбувається повернення значення, яке присвоєно зміннsq в даний час. Слід пам'ятати, що при посиланні на значення змінної знак долара повинен бути приведений, а якщо змінна вказана в цілях привласнення їй значення, то знак долара не використовується. Нижче наведено приклад, що дозволяє пояснити сказане.

**$ cat test4**

**#!/bin/bash**

**# Присвоєння значення однієї змінної іншій**

**value1=10**

**value2=$value1**

**echo The resulting value is $value2**

**Виконання математичних обчислень**

Ще одним важливим засобом будь-якої мови програмування є здатність маніпулювати числами. На жаль, в сценаріях командного інтерпретатора це завдання вирішується достатньо складно. Передбачено два способи виконання математичних операцій в сценаріях командного інтерпретатора, які розглядаються нижче.

**Команда expr**

Спочатку в командному інтерпретаторі Bourne була передбачена спеціальна команда, призначена для обробки математичних виразів. Команда expr дозволяла обробляти вираження, задані в командному рядку, але для цього застосовувалися надзвичайно складні конструкції:

**$ expr 1+5**

**6**

Команда **expr**розпізнає деякі математичні та строкові оператори, які показані в табл. 5.1.

Табл. 5.1. Оператори команди **expr**

|  |  |
| --- | --- |
| **оператор** | **опис** |
| **ARG1 | ARG2** | Повертає ARG1, якщо жоден з аргументів не має значення null або 0; в іншому випадку повертає ARG2 |
| **ARG1 & ARG2** | Повертає ARG1, якщо жоден з аргументів не має значення null або 0; в іншому випадку повертає 0 |
| **ARG1 < ARG2** | Повертає 1, якщо ARG1 менше ARG2; в іншому випадку повертає 0 |
| **ARG1 <= ARG2** | Повертає 1, якщо ARG1 менше або дорівнює ARG2; в іншому випадку повертає 0 |
| **ARG1 = ARG2** | Повертає 1, якщо ARG1 дорівнює ARG2; в іншому випадку повертає 0 |
| **ARG1! = ARG2** | Повертає 1, якщо ARG1 НЕ дорівнює ARG2; в іншому випадку повертає 0 |
| **ARG1> = ARG2** | Повертає 1, якщо ARG1 більше або дорівнює ARG2; в іншому випадку повертає 0 |
| **ARG1> ARG2** | Повертає 1, якщо ARG1 більше ARG2; в іншому випадку повертає 0 |
| **ARG1 + ARG2** | Повертає арифметичну суму ARG1 і ARG2 |
| **ARG1 - ARG2** | Повертає арифметичну різницю ARG1 і ARG2 |
| **ARG1 \* ARG2** | Повертає арифметичний добуток ARG1 і ARG2 |
| **ARG1 / ARG2** | Повертає арифметичну частку від ділення ARG1 на ARG2 |
| **ARG1% ARG2** | Повертає арифметичний залишок від ділення ARG1 на ARG2 |
| **STRING: REGEXP** | Повертає результат зіставлення з шаблоном, якщо REGEXP зіставляється з шаблоном в STRING |
| **match STRING REGEXP** | Повертає результат зіставлення з шаблоном, якщо REGEXP зіставляється з шаблоном в STRING |
| **substr STRING POS LENGTH** | Повертає підрядок довжиною LENGTH символів, починаючи з позиції POS (з відліком від 1) |
| **index STRING CHARS** | Повертає позицію в рядку STRING, в якій знайдена підрядок CHARS; в іншому випадку повертає 0 |
| **length STRING** | Повертає числове значення довжини рядка STRING |
| **+ TOKEN** | Інтерпретує TOKEN як рядок, навіть якщо це - ключове слово |
| **(EXPRESSION)** | Повертає значення EXPRESSION |

Очевидно, що ці стандартні оператори в команді **expr** діють цілком прийнятно, але при їх використанні в сценарії або в командному рядку виникають проблеми. Багато операторів команди **expr**мають в командному інтерпретаторі інші значення (як приклад можна вказати зірочку). Застосування таких операторів в команді **expr**призводить до отримання непередбачуваних результатів:

**$ expr 5\*2**

**expr: syntax error**

Для вирішення цієї проблеми доводиться використовувати символ командного інтерпретатора, що екранується, (зворотну косу риску) для позначення всіх символів, які можуть бути неправильно оброблені командним інтерпретатором, і тільки після цього передавати отримане вираз в команду **expr**:

**$ expr 5\\*2**

**10**

**Використання квадратних дужок**

Командний інтерпретатор **bash** продовжує підтримувати команду **expr**, оскільки він повинен залишатися сумісним з командним інтерпретатором Bourne. Однак в **bash** передбачений набагато більш простий спосіб виконання математичних обчислень. У командному інтерпретаторі **bash** при використанні операції присвоювання результату математичних обчислень змінної можна включити математичний вираз в конструкцію, що складається з знака долара і квадратних дужок ( **$[operation]**):

**$ var1=$[1+5]**

**$ echo $ var1**

**6**

**$ var2=$[$var1\*2]**

**$ echo $var2**

**12**

Завдяки використанню квадратних дужок проведення математичних обчислень в командному інтерпретаторі стає набагато простіше, ніж при використанні команди **expr**. Той самий метод можна застосовувати і в сценаріях командного інтерпретатора:

**$ cat test7**

**#!/bin/bash**

**var1=100**

**var2=50**

**var3=45**

**var4=$[$var1\*($var2-$var3)]**

**echo The final result is $var4**

Виконання цього сценарію призводить до отримання наступного виведення:

**$ chmod u+x test7**

**$ ./test7**

**The final result is 500**

До того ж заслуговує на увагу те, що при використанні методу на основі квадратних дужок для обчислення математичних виразів не доводиться турбуватися про те, що знак множення або будь-які інші символи будуть неправильно оброблені командним інтерпретатором. Командний інтерпретатор визначає, що даний символ не є спеціальним, оскільки він укладений у квадратні дужки.

**Підстановка результатів виконання команди**

В якості одного з найбільш корисних засобів сценаріїв командного інтерпретатора можна назвати невибагливий символ зворотної лапки, який в світі Linux зазвичай іменується зворотною одинарною лапкою (**`**). Слід завжди пам'ятати, що це - не той звичайний символ одинарної лапки, який використовується в складі рядків. Символ зворотної одинарної лапки в основному використовується лише в сценаріях командного інтерпретатора, тому багато хто навіть не уявляють собі, де цей символ знаходиться на клавіатурі. Але фахівці з Linux повинні знати, де знайти цей символ, оскільки він є вкрай важливим елементом багатьох сценаріїв командного інтерпретатора.

Символи зворотного одинарної лапки дозволяють присвоїти виведення команди командного інтерпретатора змінній. На перший погляд здається, що сказане вище не представляє собою щось особливе, але фактично ця операція є одним з важливих будівельних блоків в програмуванні сценаріїв.

У символи зворотного одинарної лапки повинна бути укладена вся команда командного рядка:

**testing=`date`**

Командний інтерпретатор виконує команду, укладену в зворотні одинарні лапки, і привласнює отриманий висновок змінної testing. Нижче наведено приклад створення змінної з використанням виведення однієї зі звичайних команд командного інтерпретатора.

**$ cat test5**

**#!/bin/bash**

**# Використання зворотної одинарної лапки**

**testing=`date`**

**echo "The date and time are: "$testing**

Мінлива **testing**отримує висновок з команди **date**, після чого отримані дані використовуються в інструкції **echo**для їх відображення. Виклик цього сценарію командного інтерпретатора на виконання призводить до отримання наступних результатів:

**$ Chmod u + x test5**

**$ ./test5**

**The date and time are: Mon Jan 31 20:23:25 EDT 2011**

**Вихід зі сценарію**

У кожній команді, виконуваної в командному інтерпретаторі, використовується так званий статус виходу для передачі командному інтерпретатору вказівки, чим закінчилася обробка цієї команди. Статус виходу - це цілочисельне значення від 0 до 255, яке передається командою командному інтерпретатору після завершення свого виконання. Це значення можна перехоплювати і використовувати в своїх сценаріях.

**Перевірка статусу виходу**

У Linux передбачена спеціальна змінна **$?**, в якій зберігається значення статусу виходу останньої виконаної команди. Значення змінної **$?**необхідно розглядати або використовувати відразу після виконання команди, статус виходу якої повинен бути перевірений. У наступному прикладі показано, як змінюється значення статусу виходу останньої команди, виконаної командним інтерпретатором:

**$ date**

**Sat Jan 15 10:01:30 EDT 2011**

**$ echo $?**

**0**

Відповідно до прийнятої угодою статус виходу успішно завершеною команди дорівнює нулю. Якщо команда завершується з помилкою, то статус виходу приймає значення позитивного цілого числа:

**$ asdfg**

**-bash: asdfg: command not found**

**$ Echo $?**

**127**

В даному випадку неприпустима команда повернула статус виходу, рівний 127. Щодо того, які значення має приймати статус виходу при виникненні тих чи інших помилок, в Linux практично немає будь-якого загальноприйнятої угоди. Однак при цьому можна керуватися деякими рекомендаціями, як показано в табл. 5.2.

Табл. 5.2. Коди статусу виходу Linux

|  |  |
| --- | --- |
| **код** | **опис** |
| **0** | Успішне завершення команди |
| **1** | Загальна невідома помилка |
| **2** | Неправильне вживання команди командного інтерпретатора |
| **126** | Команда не може бути виконана |
| **127** | Невідома команда |
| **128** | Неприпустимий аргумент виходу |
| **128+x** | Фатальна помилка з сигналом x системи Linux |
| **130** | Виконання команди завершено натисканням клавіш <Ctrl + C> |
| **255** | Значення статусу виходу не відповідає допустимому діапазону |

Значення статусу виходу, рівне 126, вказує, що для користувача не задані відповідні дозволи на виконання команди:

**$ ./myprog.c**

**-bash: ./myprog.c: Permission denied**

**$ echo $?**

**126**

Ще одна поширена помилка, з якою стикаються практично всі, полягає у використанні неприпустимого параметра для виклику команди:

**$ date %t**

**date: invalid date '% t'**

**$ echo $?**

**1**

При цьому формується загальний код статусу виходу, рівний 1, який вказує, що в команді відбулася невідома помилка.

**Команда exit**

За замовчуванням вихід зі сценарію командного інтерпретатора відбувається з установкою для нього статусу виходу останньої команди в цьому сценарії:

**$ ./test6**

**The result is 2**

**$ echo $?**

**0**

Передбачена можливість замість цього сформувати і повернути код статусу виходу, обраний самим програмістом. Для вказівки статусу виходу при завершенні сценарію застосовується команда exit:

**$ cat test13**

**#!/bin/bash**

**# Перевірка статусу виходу**

**var1=10**

**var2=30**

**var3=$[$var1+$var2]**

**echo The answer is $ var3**

**exit 5**

При перевірці статусу виходу сценарію буде отримано значення, задане в якості параметра команди **exit**:

**$ chmod u+x test13**

**$ ./test13**

**The answer is 40**

**$ echo $?**

**5**

При визначенні параметра команди **exit**можна також використовувати змінні. Однак при використанні цього засобу необхідно дотримуватися обережності, оскільки значення кодів статусу виходу не можуть перевищувати 255. Розглянемо, що відбувається в наступному прикладі:

**$ cat test14**

**#!/bin/bash**

**# Перевірка статусу виходу**

**var1=10**

**var2=30**

**var3=$[$var1\*var2]**

**echo The value is $ var3**

**exit $var3**

Після виклику на виконання даного сценарію буде отримано наступний результат:

**$ ./test14**

**The value is 300**

**$ echo $?**

**44**

Значення коду статусу виходу відсічно так, щоб він не виходив за межі діапазону від 0 до 255. У командному інтерпретаторі для здійснення цієї дії застосовується арифметика обчислення залишку від ділення (обчислення по модулю). Модуль значення - це залишок від ділення. Отримане число є залишком після поділу заданого числа на 256. У разі застосування значення 300 (в якості значення результату) залишок становить 44, і саме це значення відображається як код статусу виходу.

**Отримання введення даних від користувача**

Іноді в сценарії потрібно організувати більш тісну взаємодію з тим, хто з ним працює. Зокрема, в певні моменти виникає необхідність в ході виконання сценарію задати питання і чекати отримання відповіді від особи, що викликав сценарій на виконання. У командному інтерпретаторі **bash**якраз для цієї мети передбачена команда **read**.

Команда **read**приймає вхідні дані з файлу стандартного введення (з клавіатури) або з файлу з іншим дескриптором. Після отримання вхідних даних команда **read**поміщає їх в стандартну змінну. Нижче показаний найбільш простий варіант застосування команди **read**.

**$ cat test21**

**#!/bin/bash**

**# Перевірка команди read**

**echo -n "Enter your name:"**

**read name**

**echo "Hello $ name, welcome to my program."**

**$ ./test21**

**Enter your name: Rich Blum**

**Hello Rich Blum, welcome to my program.**

Очевидно, що тут немає нічого складного. Заслуговує на увагу те, що в команді **echo**, за допомогою якої формується запрошення до введення інформації, присутня опція **-n**. Ця опція пригнічує висновок символу позначення кінця рядка після відображення тексту самої рядки, що дозволяє користувачеві сценарію вводити дані відразу після показаної рядки запрошення, а не в наступному рядку. В результаті робота зі сценарієм стає більшою мірою схожою на роботу з формами.

Насправді команда **read**підтримує навіть опцію **-p**, яка дозволяє задавати запрошення безпосередньо в командному рядку **read**:

**$ cat test22**

**#!/bin/bash**

**# Перевірка опції read -p**

**read -p "Please enter your age:" age**

**days = $ [$ age \* 365]**

**echo "That makes you over $days days old!"**

**$ ./test22**

**Please enter your age: 10**

**That makes you over 3650 days old!**

Заслуговує на увагу те, що в першому прикладі, призначеному для введення імені, команда **read**привласнює обидва значення, ім'я та прізвище, одній і тій самій змінній. Команда **read**може привласнювати всі дані, введені в запрошенні, одній зміннй або, за бажанням, декільком змінним. В останньому випадку кожне введене значення даних присвоюється черговій змінній в списку. Якщо список змінних містить менше змінних в порівнянні з кількістю елементів даних, то все решта дані присвоюються останньої змінної:

**$ cat test23**

**#!/bin/bash**

**# Введення декількох змінних**

**read -p "Enter your name:" first last**

**echo "Checking data for $ last, $ first ..."**

**$ ./test23**

**Enter your name: Rich Blum**

**Checking data for Blum, Rich ...**

У командному рядку виклику команди **read**можна також взагалі не ставити змінні. Якщо застосовується такий варіант виклику команди **read**, то команда поміщає всі отримані нею дані в спеціальну змінну середовища **REPLY**, яка в такому випадку включає всі дані, задані в якості вхідних, і може використовуватися в сценарії командного інтерпретатора поряд з будь-якої іншою змінною.

**Лекція №6**

**«Передача параметрів командного рядка у сценарії bash та їх обробка. Команди розгалуження та цикли у сценаріях bash. Команда перевірки умов»**

**Параметри командного рядка**

Найбільш простий спосіб передачі даних в сценарій командного інтерпретатора полягає у використанні параметрів командного рядка. Параметри командного рядка дозволяють додатково задавати значення даних в командному рядку при виконанні сценарію:

**$ ./addem 10 30**

У цьому прикладі в сценарій **addem**передаються два параметри командного рядка (10 і 30). Обробка параметрів командного рядка в сценарії проводиться з використанням спеціальних змінних.

Командний інтерпретатор **bash** привласнює спеціальні змінні, що називаються позиційними параметрами, всі параметри, введені в командному рядку. У число цих параметрів включено навіть ім'я програми, виконуваної командним інтерпретатором. Позиційні змінні параметрів представляють собою стандартні числові позначення: **$0**вказує ім'я програми, **$1**відповідає першому параметру, **$2**- другому параметру і так далі, до позначення **$9**, відповідного дев'ятому параметру.

Дані змінні можна використовувати точно так само, як і будь-яку іншу змінну в сценарії командного інтерпретатора. Сценарій командного інтерпретатора автоматично привласнює змінній значення параметра командного рядка; користувач не повинен для цього виконувати будь-які дії. Параметри командного рядка слід розділяти проміжками:

**$ cat test2**

**#!/bin/bash**

**# Перевірка з двома параметрами командного рядка**

**total = $ [$ 1 \* $ 2]**

**echo The first parameter is $ 1.**

**echo The second parameter is $ 2.**

**echo The total value is $ total.**

**$ ./test2 2 5**

**The first parameter is 2.**

**The second parameter is 5.**

**The total value is 10.**

Якщо в сценарії буде потрібно більше дев'яти параметрів командного рядка, то можна продовжити їх введення, але в такому випадку імена змінних трохи зміняться. Після дев'ятої змінної необхідно ставити фігурні дужки навколо номера змінної, як в прикладі **${10}**. Нижче показано, як цей спосіб позначення змінних застосовується в сценарії.

**$ cat test4**

**#!/bin/bash**

**# Обробка великої кількості параметрів**

**total=$[${10}\*${11}]**

**echo The tenth parameter is ${10}**

**echo The eleventh parameter is ${11}**

**echo The total is $ total**

**$ ./test4 1 2 3 4 5 6 7 8 9 10 11 12**

**The tenth parameter is 10**

**The eleventh parameter is 11**

**The total is 110**

Даний метод дозволяє вводити таку кількість параметрів командного рядка для сценаріїв, що може знадобитися на практиці.

**Читання імені програми**

Для визначення імені програми, запущеної командним інтерпретатором з командного рядка, можна використовувати параметр **$0**. Необхідність в цьому виникає, наприклад, при написанні програми, яка може виконувати кілька функцій. Однак при цьому виникає невелика проблема, яку доводиться враховувати на практиці. Розглянемо, що відбувається в наступному простому прикладі:

**$ cat test5**

**#!/bin/bash**

**# Перевірка із застосуванням параметра $0**

**echo The command entered is: $0**

**$ ./test5**

**The command entered is: ./test5**

**$ /home/rich/test5**

**The command entered is: /home/rich/test5**

Якщо рядком, фактично переданим у змінну **$0**, є весь шлях до сценарію, то для вказівки на програму використовується повне позначення шляху, а не тільки ім'я самої програми.

Якщо повинен бути написаний сценарій, який виконує різні функції з урахуванням того, під яким ім'ям цей сценарій викликаний з командного рядка, то необхідно додатково виконати невелику роботу. Потрібно видалити відомості про шлях, який використовувався для виклику сценарію з командного рядка.

На щастя, передбачена невелика і зручна команда, яка призначена саме для цього. Команда **basename** повертає тільки ім'я програми без позначення шляху.

**Спеціальні змінні параметрів**

У командному інтерпретаторі **bash** передбачено кілька спеціальних змінних, які дозволяють відслідковувати параметри командного рядка.

Іноді замість перевірки кожного параметра можна просто підрахувати, скільки параметрів було введено в командному рядку. З цією метою в командному інтерпретаторі **bash** передбачена спеціальна змінна. Спеціальна змінна **$#**містить значення кількості параметрів командного рядка, заданих при виклику сценарію на виконання. Цю спеціальну змінну можна використовувати в будь-якому місці сценарію, точно так само, як і звичайну змінну:

**$ cat test8**

**#!/bin/bash**

**# Отримання числа параметрів**

**echo There were $ # parameters supplied.**

**$ ./test8**

**There were 0 parameters supplied.**

**$ ./test8 1 2 3 4 5**

**There were 5 parameters supplied.**

**$ ./test8 1 2 3 4 5 6 7 8 9 10**

**There were 10 parameters supplied.**

**$ ./test8 "Rich Blum"**

**There were 1 parameters supplied.**

У деяких ситуаціях може знадобитися просто захопити всі параметри, задані в командному рядку, і провести ітерації за всіма цими параметрами. У такому випадку замість застосування трудомісткого способу, який передбачає використання змінної **$#**для визначення кількості параметрів в командному рядку з подальшою обробкою в циклі всіх цих параметрів, можна використовувати кілька інших спеціальних змінних.

Змінні **$\***і **$@**надають зручний доступ до всіх параметрів. Обидві ці змінні включають всі параметри командного рядка у вигляді однієї змінної.

Змінна **$\***приймає всі параметри, задані в командному рядку, як одне слово. В цьому слові міститься кожне з значень в тій послідовності, в якій вони наведені в командному рядку. По суті, змінна $\* дозволяє розглядати параметри не як кілька об'єктів, а як один параметр.

З іншого боку, змінна **$@**сприймає всі параметри, задані в командному рядку, як окремі слова в одному й тому ж рядку. Вона дозволяє проводити ітерації за отриманим значенням, виділяючи кожен із заданих параметрів. Для цього найчастіше використовується команда **for**.

Однак можна легко заплутатися, намагаючись розібратися в тому, як працюють ці дві змінні. Розглянемо, яким чином можна провести між ними відмінність:

**$ Cat test11**

**#!/bin/bash**

**# Перевірка змінних $\* і $@**

**echo "Using the \$\* method: $\*"**

**echo "Using the \$ method: $@"**

**$ ./test11 rich barbara katie jessica**

**Using the $\* method: rich barbara katie jessica**

**Using the $@ method: rich barbara katie jessica**

Відзначимо, що на перший погляд при використанні обох змінних виробляється однакове виведення, яке показує відразу всі надані параметри командного рядка.

А наступний приклад показує, в чому виникають розбіжності:

**$ cat test12**

**#!/bin/bash**

**# Перевірка змінних $\* і $@**

**count = 1**

**for param in "$\*"**

**do**

**echo "\$\* Parameter # $count = $param"**

**count=$[$count+1]**

**done**

**count=1**

**for param in "$@"**

**do**

**echo "\$@ Parameter # $count = $param"**

**count=$[$count+1]**

**done**

**$ ./test12 rich barbara katie jessica**

**$ \* Parameter # 1 = rich barbara katie jessica**

**$ @ Parameter # 1 = rich**

**$ @ Parameter # 2 = barbara**

**$ @ Parameter # 3 = katie**

**$ @ Parameter # 4 = jessica**

**Застосування зсуву**

Ще одним інструментом, який надає командний інтерпретатор **bash** в розпорядження користувача, є команда **shift**. Призначення команди **shift**, підтримуваної командним інтерпретатором **bash** , полягає в тому, що вона спрощує управління параметрами командного рядка. Команда **shift**відповідно до свого смислового значення (зрушення) буквально зміщує параметри командного рядка, змінюючи їх відносні положення.

Після кожного виконання команди **shift**за замовчуванням відбувається переміщення кожної змінної параметра вниз на одну позицію. Таким чином, значення змінної **$3**переміщається в змінну **$2**, значення змінної **$2**- в змінну **$1**, а значення змінної **$1**відкидається (зверніть увагу на те, що значення змінної **$0**, яка вказує ім'я програми, залишається незмінним).

Застосування зазначеної команди є ще один зручний спосіб ітерації у командних рядках, особливо якщо неможливо дізнатися заздалегідь кількість цих параметрів. Досить просто провести обробку першого параметра, застосувати до параметрів команду зсуву, а потім знову обробити перший параметр.

Ще один варіант полягає в тому, щоб проводити зміщення на кілька позицій, задаючи в команді **shift**відповідний параметр. Досить просто вказати кількість позицій, на яке повинні бути зрушені параметри:

**$ cat test14**

**#!/bin/bash**

**# Демонстрація зсуву на кілька позицій**

**echo "The original parameters: $ \*"**

**shift 2**

**echo "Here's the new first parameter: $ 1"**

**$ ./test14 1 2 3 4 5**

**The original parameters 1 2 3 4 5**

**Here's the new first parameter: 3**

Ставлячи значення в команді **shift**, можна легко забезпечити пропуск непотрібних параметрів.

**Використання структурованих команд**

Для багатьох програм потрібно забезпечити в тому чи іншому вигляді логічне управління потоком даних, переданих між командами в сценарії. Це означає, що командний інтерпретатор повинен виконувати певні команди при одній сукупності умов, але може також переходити до обробки інших команд, якщо умови виявляться іншими. Передбачено цілий ряд команд, які дозволяють пропускати в сценарії ряд команд або обробляти команди в циклі після перевірки умов, які визначаються значеннями змінних або результатами інших команд. В цілому такі команди іменуються структурованими командами.

Структуровані команди дозволяють змінювати хід виконання інструкцій в програмі, виконуючи одні команди при одних умовах і пропускаючи інші при інших умовах. Кількість структурованих команд, передбачених в командному інтерпретаторі **bash** , дуже велике, тому будемо розглядати їх окремо.

**Інструкція if-then**

До найбільш важливих типів структурованих команд відноситься інструкція **if-then**, що має такий вигляд:

**if command**

**then**

**commands**

**fi**

Для тих, хто використовував інструкцію **if-then**в інших мовах програмування, цей формат може здатися досить складним. У багатьох інших мовах програмування об'єктом, розташованим слідом за інструкцією **if**, є вираз, значення якого обчислюється для визначення того, чи рівне воно TRUE або FALSE. Інструкція **if** командного інтерпретатора **bash** функціонує інакше.

В інструкції **if**командного інтерпретатора **bash** виконується команда **command**, задана в рядку **if**. Якщо статус виходу команди дорівнює нулю (команда завершена успішно), виконуються команди **commands**, перераховані в розділі **then**. Якщо статус виходу команди має значення, відмінне від нуля, команди в розділі then не виконуються і командний інтерпретатор **bash** переходить до наступної команди в сценарії.

Нижче наведено простий приклад, що дозволяє пояснити сказане.

**$ cat test1**

**#!/bin/bash**

**# Перевірка інструкції if**

**if date**

**then**

**echo "it worked"**

**fi**

У цьому сценарії в рядку **if**використовується команда **date**. Після успішного завершення команди інструкція echo повинна відобразити текстовий рядок. Виклик цього сценарію на виконання з командного рядка призводить до отримання наступних результатів:

**$ ./test1**

**Sat Jan 23 14:09:24 EDT 2011**

**it worked**

Командний інтерпретатор виконав команду **date**, задану в рядку **if**. Статус виходу цієї команди дорівнював нулю, тому була також виконана інструкція echo, наведена в розділі **then**.

Розглянемо ще один приклад:

**$ Cat test2**

**#!/bin/bash**

**# Перевірка неправильно заданої команди**

**if asdfg**

**then**

**echo "it did not work"**

**fi**

**echo "we are outside of the if statement"**

**$ ./test2**

**./test2: line 3: asdfg: command not found**

**we are outside of the if statement**

У цьому прикладі була навмисно використана така команда, виконання якої в рядку інструкції **if**має закінчитися невдачею. Оскільки це неправильна команда, в ній виробляється статус виходу, відмінний від нуля, і командний інтерпретатор **bash** пропускає інструкцію **echo**в розділі **then**. Слід також зазначити, що у висновку сценарію все одно з'являється повідомлення про помилку, сформований при виконанні команди в інструкції **if**.

**Інструкція if-then-else**

В інструкції **if-then**передбачена можливість лише визначити, чи успішно було виконано деяку команду. Якщо ж команда повертає ненульовий код статусу виходу, то командний інтерпретатор **bash**просто переходить до наступної команди в сценарії. У такій ситуації іноді було б зручно мати можливість виконати альтернативний набір команд. Саме для цього призначена інструкція **if-then-else**.

Інструкція **if-then-else**надає можливість задати ще одну групу команд в інструкції:

**if command**

**then**

**commands**

**else**

**commands**

**fi**

команда в рядку інструкції **if**повертає нульовий код статусу виходу, виконуються команди, перераховані в розділі **then**, як і в звичайній інструкції **if-then**. Якщо ж команда в рядку інструкції **if**повертає ненульовий код статусу виходу, командний інтерпретатор **bash** виконує команди в розділі **else**.

Тепер можна внести зміни в перевірки сценарій, який виглядає наступним чином:

**$ cat test4**

**#!/bin/bash**

**# Перевірка з використанням розділу else**

**testuser=badtest**

**if grep $testuser /etc/passwd**

**then**

**echo The files for user $ testuser are:**

**ls -a /home/$testuser/.b\***

**else**

**echo "The user name $ testuser does not exist on this system"**

**fi**

**$ ./test4**

**The user name badtest does not exist on this system**

Така конструкція є більш зручною у використанні. Розділ **else**, як і розділ **then**, може включати кілька команд. Кінець розділу **else**позначається інструкцією **fi**.

Іноді в коді сценарію необхідно перевірити одне за іншим кілька умов. Замість того щоб записувати окремі інструкції **if-then**, можна скористатися альтернативним варіантом розділу **else**, який позначається як **elif**.

Розділ **elif**продовжує розділ **else**, задаючи ще одну інструкцію **if-then**:

**if command1**

**then**

**commands**

**elif command2**

**then**

**more commands**

**fi**

У рядку інструкції **elif**міститься інша команда, яка підлягає виконанню, однак вона має те ж призначення, що і в рядку вихідної інструкції **if**. Якщо код статусу виходу команди **elif**дорівнює нулю, командний інтерпретатор **bash** виконує команди в другому розділі інструкції **then**.

Інструкції **elif**можна продовжувати пов'язувати в ланцюжки, що іноді призводить до створення величезних конгломератів інструкцій **if-then-elif**:

**if command1**

**then**

**command set 1**

**elif command2**

**then**

**command set 2**

**elif command3**

**then**

**command set 3**

**elif command4**

**then**

**command set 4**

**fi**

**Команда test**

До сих пір в цьому розділі розглядалося застосування в рядку інструкції **if**лише звичайних команд командного інтерпретатора. Може виникнути питання, чи володіє інструкція **if-then**командного інтерпретатора **bash** здатністю оцінювати якісь інші умови, крім коду статусу виходу команди.

Відповідь є негативною - не може. Однак у складі командного інтерпретатора **bash** є зручна програма, що дозволяє перевіряти інші умови і призначена для використання в інструкції **if-then**.

Один із способів перевірки різних умов в інструкції if-then якраз і полягає в використанні команди **test**. Якщо перевірка умови, заданого в команді **test**, призводить до отримання істинного значення, то команда test виконує повернення з нульовим кодом статусу виходу. Таким чином, виконання інструкції **if-then**більшою мірою нагадує дію інструкції **if-then**в інших мовах програмування. Якщо перевірка умови призводить до отримання хибного значення, команда **test**повертає 1, тому команди в розділі **then**інструкції **if-then**пропускаються.

Команда **test**має досить простий формат:

**test condition**

Умова **condition**- це ряд параметрів і значень, що обробляються командою **test**. При використанні в інструкції **if-then**команда **test**виглядає наступним чином:

**if test condition**

**then**

**commands**

**fi**

У командному інтерпретаторі **bash** передбачений альтернативний спосіб включення команди **test**в інструкцію **if-then**:

**if [ condition ]**

**then**

**commands**

**fi**

Квадратні дужки визначають умова, що використовується в команді **test**. При використанні цієї команди необхідно дотримуватися обережності; повинен бути обов'язково введений проміжок після першої квадратної дужки і перед останньою квадратною дужкою, оскільки в іншому випадку буде отримано повідомлення про помилку.

Команда **test**може обробляти умови трьох типів:

1. порівняння чисел;
2. порівняння рядків;
3. порівняння файлів.

**Порівняння чисел**

Найбільш широко застосовуваний метод роботи з командою **test**полягає у використанні цієї команди для порівняння двох числових значень. У табл. 10.1 наведено список параметрів умов, які можуть служити для перевірки двох значень.

Табл. 6.1. Операції порівняння чисел в команді **test**

|  |  |
| --- | --- |
| **порівняння** | **опис** |
| **n1 -eq n2** | Перевірка того, що **n1**одно **n2** |
| **n1 -ge n2** | Перевірка того, що **n1**більше або дорівнює **n2** |
| **n1 -gt n2** | Перевірка того, що **n1**більше **n2** |
| **n1 -le n2** | Перевірка того, що **n1**менше або дорівнює **n2** |
| **n1 -lt n2** | Перевірка того, що **n1**менше **n2** |
| **n1 -ne n2** | Перевірка того, що **n1**не дорівнює **n2** |

Умови перевірки числових значень можуть використовуватися для роботи і з числами, і з змінними. Нижче наведено приклад застосування такої перевірки.

**$ cat test5**

**#!/bin/bash**

**# Використання операторів перевірки числових значень**

**val1=10**

**val2=11**

**if [ $val1 -gt 5 ]**

**then**

**echo "The test value $ val1 is greater than 5"**

**fi**

**if [ $val1 -eq $val2 ]**

**then**

**echo "The values ​​are equal"**

**else**

**echo "The values ​​are different"**

**fi**

**Порівняння рядків**

Команда **test**дозволяє також виконувати порівняння строкових значень. Оператори порівняння, які можуть використовуватися для обробки двох строкових значень, наведені в табл. 6.2.

Табл. 6.2. Оператори порівняння рядків в команді **test**

|  |  |
| --- | --- |
| **порівняння** | **опис** |
| **str1 = str2** | Перевірка того, що рядок **str1**є таким же, як і рядок **str2** |
| **str1! = str2** | Перевірка того, що рядок **str1**не збігається з рядком **str2** |
| **str1 < str2** | Перевірка того, що рядок **str1**менше рядка **str2** |
| **str1 > str2** | Перевірка того, що рядок **str1**більше рядка **str2** |
| **-n str1** | Перевірка того, що рядок **str1**має довжину більше нуля |
| **-z str1** | Перевірка того, що рядок **str1**має довжину, рівну нулю |

Стосовно до рядків перевірка на рівність і нерівність є цілком очевидний процес. Визначити, чи є два строкових значення однаковими або неоднаковими, не складає особливих труднощів:

**$ cat test7**

**#!/bin/bash**

**# Перевірка рядків на рівність**

**testuser=rich**

**if [ $USER = $testuser ]**

**then**

**echo "Welcome $testuser"**

**fi**

**$ ./test7**

**Welcome rich**

Крім того, порівняння рядків на нерівність також дозволяє визначити, чи мають два рядки одне і те ж значення чи ні.

При порівнянні рядків за допомогою команди **test** враховуються всі знаки пунктуації і відмінності в застосуванні великих і малих літер в ході того, як відбувається перевірка рядків на рівність.

Завдання ускладнюється при здійсненні спроби визначити, чи повинен один рядок розглядатися як попередній іншому рядку в послідовності сортування по зростанню. Програмісти на мові командного інтерпретатора часто стикаються з наступними двома підступними проблемами при спробі використовувати операції "більше" або "менше" стосовно до рядків в команді **test**.

* По-перше, символи " **>**" і " **<**" повинні бути екрановані, оскільки в іншому випадку в командному інтерпретаторі ці символи будуть розглядатися як символи перенаправлення, а строкові значення - як імена файлів.
* По-друге, рядки, впорядковують за допомогою операторів " **>**" і " **<**", розташовуються інакше, ніж відсортовані за допомогою команди **sort**.

**Порівняння файлів**

Команда **test**дозволяє перевіряти статус файлів і каталогів в файлової системі Linux. Застосовувані при цьому оператори порівняння перераховані в табл. 6.3.

Табл. 6.3. Порівняння файлів в команді **test**

|  |  |
| --- | --- |
| **порівняння** | **опис** |
| **-d file** | Перевірка того, що **file**існує і є каталогом |
| **-e file** | Перевірка того, що **file**існує |
| **-f file** | Перевірка того, що **file**існує і є файлом |
| **-r file** | Перевірка того, що **file**існує і призначений для читання |
| **-s file** | Перевірка того, що **file**існує і не порожній |
| **-w file** | Перевірка того, що **file**існує і призначений для запису |
| **-x file** | Перевірка того, що **file**існує і призначений для виконання |
| **-O file** | Перевірка того, що **file**існує і належить поточному користувачеві |
| **-G file** | Перевірка того, що **file**існує і його задана за замовчуванням група є такою ж, як і у поточного користувача |
| **file1 -nt file2** | Перевірка того, що **file1**- новіший, ніж **file2** |
| **file1 -ot file2** | Перевірка того, що **file1**- старіший, ніж **file2** |

Ці умови надають можливість перевіряти в сценаріях командного інтерпретатора файли в файлової системі і часто використовуються в сценаріях, заснованих на доступі до файлів. Оскільки дані перевірки знаходять настільки широке застосування, розглянемо кожну з них окремо.

**Приклад 1.**

**$ cat test11**

**#!/bin/bash**

**# Перевірка перед переходом**

**if [ -d $HOME ]**

**then**

**echo "Your HOME directory exists"**

**cd $ HOME**

**ls -a**

**else**

**echo "There is a problem with your HOME directory"**

**fi**

**Приклад 2**.

**$ cat test17**

**#!/bin/bash**

**# Перевірка можливості виконання файлу**

**if [ -x test16 ]**

**then**

**echo "You can run the script:"**

**./test16**

**else**

**echo "Sorry, you are unable to execute the script"**

**fi**

**Приклад3.**

**$ cat test18**

**#!/bin/bash**

**# Перевірка приналежності файлу**

**if [ -O /etc/passwd ]**

**then**

**echo "You are the owner of the /etc/passwd file"**

**else**

**echo "Sorry, you are not the owner of the /etc/passwd file"**

**fi**

**Команда case**

При написанні сценаріїв часто виявляється, що необхідно обчислити значення змінної, а потім провести пошук конкретного значення серед цілого ряду можливих значень.

Для цього достатньо скористатися командою **case**. У команді **case**перевірка значення однієї змінної на відповідність декільком різним значенням організована із застосуванням формату списку:

**case variable in**

**pattern1 | pattern2) commands1 ;;**

**pattern3) commands2 ;;**

**\*) Default commands ;;**

**esac**

Команда **case**порівнює задану змінну **variable** з декількома різними шаблонами. При виявленні узгодження змінної з будь-яким з шаблонів **pattern**командний інтерпретатор виконує команди **commands**, зазначені для цього шаблону. В одному рядку можна задавати декілька шаблонів, використовуючи операцію зі знаком операції "**|**". Для відділення одного шаблону від іншого. Символ зірочки служить універсальної пасткою для тих значень, які були зіставлені з жодним з перерахованих шаблонів. У прикладі, наведеному нижче, показано використання команди **case**.

**$ cat test26**

**#!/bin/bash**

**# Використання команди case**

**case $USER in**

**rich | barbara)**

**echo "Welcome, $USER"**

**echo "Please enjoy your visit" ;;**

**testing)**

**echo "Special testing account" ;;**

**jessica)**

**echo "Do not forget to log off when you're done" ;;**

**\*)**

**echo "Sorry, you are not allowed here" ;;**

**esac**

**$ ./test26**

**Welcome, rich**

**Please enjoy your visit**

**Цикли**

**for**

У практиці програмування широко застосовується спосіб організації роботи шляхом виконання ітерацій з охопленням ряду команд. Часто виникає необхідність повторювати ряд команд до тих пір, поки не буде виконано якусь конкретну умова, наприклад, будуть опрацьовані всі файли в каталозі, дані по всім користувачам в системі або всі рядки в текстовому файлі.

Командний інтерпретатор **bash** надає можливість використовувати команду **for** для створення циклу, в якому проводиться ітерація по ряду значень. У кожній ітерації виконується певний набір команд з використанням одного зі значень у ряді. Нижче наведено основний формат команди **for**командного інтерпретатора **bash** .

**for var in list**

**do**

**commands**

**done**

Ряд значень, що використовується в ітераціях, задається за допомогою параметра **list**. Передбачено кілька різних способів завдання значень в цьому списку.

У кожній ітерації змінна **var**містить поточне значення в списку **list**. При першій ітерації використовується перший елемент у списку, при другій ітерації - другий елемент і так далі, до вичерпання всіх елементів в списку.

Щодо команд **commands**, які вводяться між інструкціями **do** і **done**, можна задавати одну або кілька стандартних команд командного інтерпретатора **bash**. При виконанні команд може використовуватися змінна **$var**, яка містить поточне значення елемента списку для даної ітерації.

Можна здійснити кількома способами завдання значень в списку.

1. **Читання значень в списку**

Найбільш простий спосіб використання команди **for**складається в здійсненні з її допомогою ітерацій за списком значень, визначеному в самій команді **for**:

**$ cat test1**

**#!/bin/bash**

**# Команда for в основній формі**

**for test in Alabama Alaska Arizona Arkansas California Colorado**

**do**

**echo The next state is$ test**

**done**

**$ ./test1**

**The next state is Alabama**

**The next state is Alaska**

**The next state is Arizona**

**The next state is Arkansas**

**The next state is California**

**The next state is Colorado**

1. **Читання списку зі змінної**

При програмуванні на мові сценаріїв командного інтерпретатора часто доводиться стикатися з такою ситуацією, коли накопичується список значень і зберігається у змінній, після чого повинні бути виконані ітерації за елементами списку. Це також можна зробити за допомогою команди **for**:

**$ cat test4**

**#!/bin/bash**

**# Використання змінної для зберігання списку**

**list="Alabama Alaska Arizona Arkansas Colorado"**

**list=$list"Connecticut"**

**for state in $list**

**do**

**echo "Have you ever visited $ state?"**

**done**

**$ ./test4**

**Have you ever visited Alabama?**

**Have you ever visited Alaska?**

**Have you ever visited Arizona?**

**Have you ever visited Arkansas?**

**Have you ever visited Colorado?**

**Have you ever visited Connecticut?**

Змінна **$list** містить стандартний список текстових значень, які повинні використовуватися для здійснення ітерацій. Заслуговує на увагу те, що в цьому коді використовується також ще один оператор присвоювання для додавання (або конкатенації) елемента до існуючого списку, що міститься в змінній **$list**. Це - загальноприйнятий метод додавання тексту до кінця існуючого текстового рядка, що зберігається у змінній.

1. **Читання значень з команди**

Ще один спосіб формування значень для використання в списку полягає в застосуванні виведення команди. При цьому використовуються символи зворотної одинарної лапки для виконання будь-якої команди, що виробляє виведення, яке потім застосовується в команді **for**:

**$ cat test5**

**#!/bin/bash**

**# Читання значень з файлу**

**file="states"**

**for state in 'cat $ file'**

**do**

**echo "Visit beautiful $ state"**

**done**

**$ cat states**

**Alabama**

**Alaska**

**Arizona**

**Arkansas**

**Colorado**

**Connecticut**

**Delaware**

**Florida**

**Georgia**

**$ ./test5**

**Visit beautiful Alabama**

**Visit beautiful Alaska**

**Visit beautiful Arizona**

**Visit beautiful Arkansas**

**Visit beautiful Colorado**

**Visit beautiful Connecticut**

**Visit beautiful Delaware**

**Visit beautiful Florida**

**Visit beautiful Georgia**

У цьому прикладі команда **cat**, задана в зворотних одинарних лапках, служить для відображення даних про стан файлів. Заслуговує на увагу те, що файл з даними про стани має такий формат: кожний стан приведений в окремому рядку, а не заданий в загальному рядку з поділом проміжками. Команда **for** як і раніше виконує ітерації з виведення команди **cat**, обробляючи кожен раз один рядок, виходячи з припущення, що кожний стан представлено в окремому рядку. Однак це не вирішує проблему наявності прогалин в даних. Якщо в списку знаходяться дані про стан, що містять проміжки, то в команді **for**все одно кожен елемент даних, відокремлених від інших пробілами, розглядається як окреме значення.

**Команда for в стилі мови C**

У мові C в інструкції **for**передбачені: певний спосіб вказівки змінної, спосіб завдання умови, яка повинно залишатися істинною, щоб тривали ітерації, і спосіб зміни значення змінної при кожній ітерації. Якщо зазначена умова приймає помилкове значення, цикл **for** зупиняється. Вираз умови визначається з використанням стандартних математичних символів. Як приклад розглянемо наступний код на мові C:

**for (i = 0; i <10; i ++)**

**{**

**printf ( "The next number is% d \ n", i);**

**}**

При виконанні цього коду здійснюється простий ітеративний цикл, в якому змінна i використовується як лічильник. У першому розділі інструкції цієї змінної присвоюється значення за замовчуванням. У другому розділі визначається умова, при виконанні якого ітерації в циклі тривають. Якщо задана умова приймає помилкове значення, ітерації в циклі **for**припиняються. У третьому, останньому розділі інструкції визначається ітеративний процес. Після кожної ітерації виконується вираз, визначене в останньому розділі. У цьому прикладі змінна i після кожної ітерації збільшується на одиницю.

У командному інтерпретаторі **bash** також підтримується версія циклу **for**, яка виглядає аналогічно циклу **for**у мові C, хоча і має деякі тонкі відмінності, в тому числі такі важливі, які можуть стати причиною плутанини в роботі програмістів на мові сценаріїв командного інтерпретатора. Нижче наведено основний формат циклу for командного інтерпретатора **bash** в стилі мови C.

**for ((variable assignment; condition; iteration process))**

Формат циклу **for**в стилі мови C може стати причиною плутанини для програмістів на мові сценаріїв командного інтерпретатора **bash**, оскільки в ньому використовуються посилання на змінні в стилі C замість посилань на змінні в стилі командного інтерпретатора. Нижче показано, як виглядає команда **for** в стилі C.

**for ((a = 1; a <10; a ++))**

Відзначимо кілька особливостей цієї команди, які характеризують її відмінність від стандартної інструкції **for** командного інтерпретатора **bash** .

* Інструкція надання значення змінної може містити проміжки;
* Змінній в умові не передує знак долара;
* Вираз, за допомогою якого організовується ітеративний процес, не відповідає формату команди **expr**.

Цей формат був створений розробниками командного інтерпретатора так, щоб він найбільшою мірою відповідав команді **for**в стилі мови C. Зрозуміло, це дозволяє спростити роботу програмістів, які володіють мовою C, а що стосується програмістів на мові сценаріїв командного інтерпретатора, то навіть найбільш досвідчені з них можуть відчути себе розгубленими. Будьте обережні при використанні циклу **for**в стилі C в конкретних сценаріях.

Нижче наведено приклад застосування команди for в стилі C в програмі командного інтерпретатора **bash**.

**$ cat test8**

**#!/bin/bash**

**# Перевірка в циклі for в стилі C**

**for ((i=1; i<=10; i++))**

**do**

**echo "The next number is $ i"**

**done**

**$ ./test8**

**The next number is 1**

**The next number is 2**

**The next number is 3**

**The next number is 4**

**The next number is 5**

**The next number is 6**

**The next number is 7**

**The next number is 8**

**The next number is 9**

**The next number is 10**

У циклі **for**здійснюється ітерація по ряду команд з використанням змінної, визначеної в циклі **for**(в даному випадку змінна має ім'я **i**). У кожній ітерації змінна **$i**містить значення, присвоєне в циклі **for**. А після кожної ітерації над змінною виконується дія, передбачена в циклі; в даному прикладі відбувається збільшення значення змінної на одиницю.

**while**

Команда **while** є свого роду поєднанням інструкції **if-then** і циклу **for**. Команда **while**дозволяє визначити команду, що перевіряється, а потім обробляти в циклі ряд команд до тих пір, поки задана команда **test** повертає нульовий статус виходу. Перевірка повертається команди **test** відбувається на початку кожної ітерації. Як тільки команда **test**поверне ненульовий статус виходу, виконання ряду команд в команді **while**припиняється.

Команда **while**має такий вигляд:

**while test**

**command**

**do**

**other commands**

**done**

Команда перевірки **test**, відповідна в команді **while**, має такий самий формат, як і в інструкціях **if-then**, де можна використовувати будь-яку звичайну команду командного інтерпретатора **bash** або включити команду **test**для перевірки умов, таких як значення змінних.

Ключовою особливістю команди **while**є те, що в ній повинен змінитися статус виходу зазначеної команди перевірки в результаті виконання команд, заданих в циклі. Якщо не буде змінюватися статус виходу, цикл **while**може стати нескінченним.

Найчастіше команда перевірки застосовується в формі, яка передбачає використання квадратних дужок для перевірки значення однієї із змінних командного інтерпретатора, яка застосовується в командах циклу:

**$ Cat test10**

**#!/bin/bash**

**# Перевірка в команді while**

**var1=10**

**while [ $ var1 -gt 0 ]**

**do**

**echo $ var1**

**var1=$[$var1-1]**

**done**

**$ ./test10**

**10**

**9**

**8**

**7**

**6**

**5**

**4**

**3**

**2**

**1**

**until**

Команда **until** діє прямо протилежним способом в порівнянні з командою **while**. Команда **until**вимагає, щоб була задана команда перевірки, яка за звичайних обставин виробляє ненульовий статус виходу. До тих пір поки статус виходу команди перевірки залишається ненульовим, командний інтерпретатор **bash** виконує команди, які входять до складу циклу. Як тільки команда перевірки повертає нульовий статус виходу, цикл зупиняється.

Як і слід було очікувати, команда **until** має такий вигляд:

**until test**

**commands**

**do**

**other commands**

**done**

Аналогічно команді **while**, в інструкції команди **until** можна задавати декілька команд перевірки **test commands**. Тільки статус виходу останньої команди визначає, чи буде командний інтерпретатор **bash** виконувати інші команди **other commands**, які становлять цикл.

Приклад використання команди **until**наведено нижче.

**$ cat test12**

**#!/bin/bash**

**var1=100**

**until [ $ var1 -eq 0 ]**

**do**

**echo $var1**

**var1=$[$var1-25]**

**done**

**$ ./test12**

**100**

**75**

**50**

**25**

**Управління циклом**

Вперше приступаючи до вивчення тематики, пов'язаної з циклами, багато хто вважає, що після запуску циклу доводиться терпляче чекати, поки цикл повністю не завершить всі свої ітерації, незалежно від обставин, що склалися. Але справи йдуть інакше. Передбачено кілька команд, в тому числі зазначених нижче, які допомагають керувати тим, що відбувається в циклі.

* команда **break**;
* команда **continue**.

Кожна з цих команд має різне призначення, коли справа стосується управління роботою циклу.

**break**

Команда **break** надає простий спосіб виходу з циклу в ході його роботи. Команду **break** можна використовувати для виходу з циклу будь-якого типу, включаючи цикли **until** і **while**.

При виконанні команди **break**командний інтерпретатор робить спробу вийти з циклу, який обробляється в даний час. Якщо робота програми організована за принципом застосування декількох циклів, то команда **break**автоматично завершує самий внутрішній виконується цикл.

Іноді при виконанні внутрішнього циклу виникає необхідність зупинити зовнішній цикл. Для цього в команді break передбачений окремий параметр командного рядка:

**break n**

де **n**вказує рівень циклу, з якого повинен бути виконаний вихід. За замовчуванням **n**дорівнює 1, а це означає, що вихід повинен бути виконаний з поточного циклу. Якщо буде встановлено значення **n**, що дорівнює 2, то команда **break** зупинить виконання зовнішнього циклу наступного рівня:

**$ cat test20**

**#!/bin/bash**

**# Вихід із зовнішнього циклу**

**for ((a=1; a<4; a++))**

**do**

**echo "Outer loop: $a"**

**for ((b=1; b<100; b++))**

**do**

**if [ $b -gt 4 ]**

**then**

**break 2**

**fi**

**echo "Inner loop: $ b"**

**done**

**done**

**$ ./test20**

**Outer loop: 1**

**Inner loop: 1**

**Inner loop: 2**

**Inner loop: 3**

**Inner loop: 4**

**Continue**

Команда **continue** надає спосіб передчасного припинення обробки частини команд в циклі без повного завершення циклу. Ця команда дозволяє задавати в циклі умови, при яких командний інтерпретатор не виконуватиме деякі команди.

Нижче наведено простий приклад використання команди continue в циклі **for**.

**$ cat test21**

**#!/bin/bash**

**# Застосування команди continue**

**for ((var1=1; var1<15; var1++))**

**do**

**if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]**

**then**

**continue**

**fi**

**echo "Iteration number: $var1"**

**done**

**$ ./test21**

**Iteration number: 1**

**Iteration number: 2**

**Iteration number: 3**

**Iteration number: 4**

**Iteration number: 5**

**Iteration number: 10**

**Iteration number: 11**

**Iteration number: 12**

**Iteration number: 13**

**Iteration number: 14**

Як і команда **break**, команда **continue**дозволяє вказувати, на якому рівні циклу повинен відбуватися пропуск команд, що залишилися, за допомогою параметра командного рядка:

**continue n**

де **n**визначає рівень циклу, на якому має відбуватися продовження роботи циклу без виконання решти операторів.

**Лекція №7**

**«Програмування функцій у сценаріях bash»**

**Створення функцій**

У міру того як розробляються сценарії командного інтерпретатора стають все більш складними, доводиться стикатися з тим, що частини коду, які виконують певні завдання, повинні знову і знову повторюватися в одному і тому ж сценарії. Іноді це достатньо прості фрагменти коду, допустимо, за допомогою яких відображається текстове повідомлення і запитується відповідь від користувача сценарію. Але зустрічаються і такі ситуації, коли в певних частинах коду виконується складне обчислення, яке повинно проводитися кілька разів на сценарії в складі більш загального процесу.

У кожній з таких ситуацій доводиться займатися стомлюючою роботою, знову і знову вводячи одні і ті ж блоки коду в текст сценарію. Тому свого часу розробники прийшли до думки, що було б зручно, якби можна було записати блок програми один раз, а потім звертатися до цього блоку коду в будь-якому місці сценарію, не повторюючи його буквально.

У командному інтерпретаторі **bash** передбачені засоби, що дозволяють зробити саме це. Одним з таких засобів є функції - блоки коду сценарію, яким присвоюється ім'я, після чого з'являється можливість викликати ці блоки в сценарії на ім'я там, де це буде потрібно. У будь-який час при виникненні необхідності скористатися заздалегідь підготовленим блоком коду в сценарії досить ввести ім'я функції, присвоєне блоку коду (ця операція називається викликом функції). У цьому розділі пояснено, як створюються і використовуються функції в сценаріях командного інтерпретатора.

Для створення функції в сценаріях командного інтерпретатора **bash** можна скористатися одним з двох форматів. У першому форматі використовується ключове слово **function** поряд з ім'ям функції, яким позначається блок коду:

**function name**

**{**

**commands**

**}**

Атрибут **name**визначає унікальне ім'я, присвоєне функції. Кожна функція, що визначається в сценарії, повинна отримати унікальне ім'я.

У цьому визначенні **commands**- це одна або кілька команд командного інтерпретатора **bash**, з яких складається дана функція. Після виклику функції командний інтерпретатор **bash**виконує кожну з цих команд в тому порядку, в якому вони присутні у визначенні функції, точно так же, як і при виконанні звичайного коду сценарію.

Другий формат визначення функції в сценарії командного інтерпретатора **bash** більшою мірою нагадує формат, який застосовується для визначення функцій в інших мовах програмування:

**name ()**

**{**

**commands**

**}**

Порожні круглі дужки після імені функції вказують на те, що далі йде визначення функції. На цей формат поширюються такі ж правила іменування функцій, як і на вихідний формат функцій сценаріїв командного інтерпретатора.

Щоб скористатися функцією в сценарії, необхідно вказати в рядку коду ім'я функції, за таким же принципом, як відбувається виклик будь-якої іншої команди командного інтерпретатора:

**$ cat test1**

**#!/bin/bash**

**# Використання функції в сценарії**

**function func1**

**{**

**echo "This is an example of a function"**

**}**

**count = 1**

**while [ $count -le 5 ]**

**do**

**func1**

**count=$[$count+1]**

**done**

**echo "This is the end of the loop"**

**func1**

**echo "Now this is the end of the script"**

**$ ./test1**

**This is an example of a function**

**This is an example of a function**

**This is an example of a function**

**This is an example of a function**

**This is an example of a function**

**This is the end of the loop**

**This is an example of a function**

**Now this is the end of the script**

В даному випадку при кожному посиланні на ім'я функції **func1** командний інтерпретатор **bash** звертається до визначення **func1 function**і виконує всі команди, передбачені в цьому визначенні.

Визначення функції не обов'язково має бути приведене у сценарії командного інтерпретатора в першу чергу, однак необхідно дотримуватися обережності. При спробі використовувати функцію до її визначення з'являється повідомлення про помилку.

Необхідно також дотримуватися певних правил, що стосуються імен функцій. Нагадаємо, що кожне ім'я функції має бути унікальним, оскільки в іншому випадку виникає проблема. Якщо в сценарії зустрічається ще одне визначення однієї і тієї ж функції, то відбувається так зване перевизначення, і нова версія функції перекриває її вихідну версію, не викликаючи жодних повідомлень про помилку

**Повернення значення з функції**

У командному інтерпретаторі **bash** функції розглядаються як свого роду міні-сценарії, зі своїм статусом виходу. Передбачено три способи, за допомогою яких можна формувати статус виходу для користувача функцій.

1. **Статус виходу, заданий за замовчуванням**

За замовчуванням статус виходу функції визначається як статус виходу, повернутий останньою командою в функції. Після завершення виконання функції можна скористатися стандартною змінною **$?**для визначення статусу виходу функції:

**$ cat test4**

**#!/bin/bash**

**# Перевірка статусу виходу функції**

**func1 ()**

**{**

**echo "trying to display a non-existent file"**

**ls -l badfile**

**}**

**echo "testing the function:"**

**func1**

**echo "The exit status is: $?"**

**$ ./test4**

**testing the function:**

**trying to display a non-existent file**

**ls: badfile: No such file or directory**

**The exit status is: 1**

В даному випадку статус виходу функції дорівнює 1, оскільки виконання останньої команди в функції закінчилося невдачею. Але це не дозволяє дізнатися, закінчилося чи виконання всіх інших команд у функції успішно чи ні. Розглянемо наступний приклад:

**$ cat test4a**

**#!/bin/bash**

**# Перевірка статусу виходу функції**

**func1 ()**

**{**

**ls -l badfile**

**echo "This was a test of a bad command"**

**}**

**echo "testing the function:"**

**func1**

**echo "The exit status is: $?"**

**$ ./test4a**

**testing the function:**

**ls: badfile: No such file or directory**

**This was a test of a bad command**

**The exit status is: 0**

Цього разу останньої інструкцією до функцій був виклик команди **echo**, який завершився успішно, тому функція має статус виходу 0, незважаючи на те, що виклик однієї з команд в функції закінчився невдачею. Таким чином, підхід, який передбачає використання заданого за замовчуванням статусу виходу функції, не завжди виправданий. На щастя, передбачені інші способи формування статусу виходу функції.

1. **Використання команди return**

У командному інтерпретаторі **bash**, щоб скасувати з конкретним статусом виходу може застосовуватися команда **retur**. Команда **return**дозволяє задати одне цілочисельне значення для визначення статусу виходу функції, що може служити простим способом завдання статусу виходу функції програмним шляхом:

**$ cat test5**

**#!/bin/bash**

**# Використання команди return в функції**

**function dbl**

**{**

**read -p "Enter a value:" value**

**echo "doubling the value"**

**return $ [$ value \* 2]**

**}**

**dbl**

**echo "The new value is $?"**

Функція **dbl**подвоює значення, що міститься у змінній **$value**, отримане в результаті введення даних користувачем. Потім в цій функції відбувається повернення сформованого результату за допомогою команди **return**, а повернене значення відображається в сценарії з використанням змінної **$?** .

Однак при використанні даного способу повернення значення з функції слід дотримуватися обережності. Щоб уникнути проблем, необхідно керуватися двома рекомендаціями:

* обов'язково виконувати вибірку повертається відразу після завершення функції;
* не забувати про те, що статус виходу повинен знаходитися в межах від 0 до 255.

Якщо перед отриманням значення функції за допомогою змінної **$?** буде виконана будь-яка інша команда, то повертається значення функції виявиться втраченим. Слід пам'ятати, що змінна **$?** повертає статус виходу останньої виконаної команди.

Додаткові обмеження на використання цього способу повернення значення накладає ще одна проблема. Статус виходу повинен бути менше 256, тому результат виконання функції повинен являти собою цілочисельне значення, що не перевищує 256. Спроба повернути будь-який більш високе значення призводить до виникнення помилки:

**$ ./test5**

**Enter a value: 200**

**doubling the value**

**The new value is 1**

Таким чином, даний спосіб повернення значення не може застосовуватися, якщо необхідно виконати повернення з функціонально, припустимо, більшого цілочисельного значення або строкового значення.

1. **Використання виведення з функції**

За аналогією з тим, що можна перехоплювати виведення команди за допомогою змінної командного інтерпретатора, можна також перехоплювати за допомогою змінної командного інтерпретатора висновок функції. Цей спосіб може використовуватися для отримання висновку будь-якого типу з функції для присвоювання його змінної:

**result='dbl'**

Ця команда присвоює виведення функції **dbl** змінній командного інтерпретатора **$result**. Нижче наведено приклад використання даного способу в сценарії.

**$ cat test5b**

**#!/bin/bash**

**# Використання команди echo для повернення значення**

**function dbl**

**{**

**read -p "Enter a value:" value**

**echo $[$value\*2]**

**}**

**result=`dbl`**

**echo "The new value is $ result"**

**$ ./test5b**

**Enter a value: 200**

**The new value is 400**

**$ ./test5b**

**Enter a value 1000**

**The new value is 2000**

У новій версії функції тепер застосовується інструкція **echo** для відображення результату обчислення. У сценарії просто відбувається перехоплення виведення функції **dbl**, а не перегляд статусу виходу для отримання відповіді.

У цьому прикладі демонструється ще один тонкий прийом. Зверніть увагу на те, що функція **dbl**фактично виводить два повідомлення. Команда **read**виводить коротке повідомлення, запитуючи у користувача подвоєне значення. Але алгоритми роботи засобів підтримки сценаріїв командного інтерпретатора **bash** виявляються достатньо інтелектуальними, щоб це повідомлення не розглядалося як частина виведення **STDOUT**, тому воно пропускається.

Якби для формування цього повідомлення із запитом користувачу застосовувалася інструкція **echo**, то повідомлення було б перехоплено і збережено в змінній командного інтерпретатора поряд з вихідним значенням.

**Передача параметрів в функцію**

Так як в командному інтерпретаторі **bash**функція розглядається як свого роду міні-сценарій, це зокрема означає, що в функцію можна передавати параметри, як і в звичайний сцнарій.

У функціях можна використовувати стандартні змінні середовища параметрів для подання будь-яких параметрів, переданих у функцію в командному рядку. Наприклад, ім'я функції визначено в змінній **$0**, а всі параметри в командному рядку функції визначаються з використанням змінних **$1**, **$2**і т. д. Крім того, для визначення кількості параметрів, що передаються в функцію, можна використовувати спеціальну змінну **$#**.

Ставлячи функцію в сценарії, необхідно приводити параметри в тому ж командному рядку, як і функцію, приблизно так:

**func1 $value1 10**

Потім в функції можна здійснити вибірку значень параметрів з використанням змінних середовища параметрів. Нижче наведено приклад застосування даного способу для передачі значень в функцію.

**$ cat test6**

**#!/bin/bash**

**# Передача параметрів в функцію**

**function addem**

**{**

**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=`addem 10 15`**

**echo $value**

**echo -n "Let's try adding just one number:"**

**value=`addem 10`**

**echo $value**

**echo -n "Now trying adding no numbers:"**

**value=`addem`**

**echo $value**

**echo -n "Finally, try adding three numbers:"**

**value=`addem 10 15 20`**

**echo $value**

**$ ./test6**

**Adding 10 and 15: 25**

**Let's try adding just one number: 20**

**Now trying adding no numbers: -1**

**Finally, try adding three numbers: -1**

У сценарії **text6**функція **addem**спочатку перевіряє кількість параметрів, переданих зі сценарію. Якщо параметри не задані або кількість параметрів більше двох, функція **addem** повертає значення -1. Якщо заданий тільки один параметр, функція **addem**складає значення цього параметра з самим собою для вироблення результату. Якщо задані два параметра, функція **addem**для формування результату складає отримані значення.

**Обробка змінних у функції**

Програмісти на мові сценаріїв командного інтерпретатора стикаються ще з одним поняттям, неправильне тлумачення якого може викликати проблему. Йдеться про область визначення змінної. Областю визначення називають ту ділянку коду, в якому змінна є видимою. Змінні, визначені у функціях, можуть мати іншу область визначення в порівнянні зі звичайними змінними. Таким чином, вони можуть бути приховані від іншої частини сценарію.

У функціях використовуються змінні двох типів:

* глобальні змінні;
* локальні змінні.

**Глобальні змінні**

Глобальні змінні - це змінні, які залишаються дійсними в будь-якому місці сценарію командного інтерпретатора. Якщо глобальна змінна визначена в основній частині сценарію, то її значення можна отримати і в функції. Аналогічним чином до значення глобальної змінної, визначеної в функції, можна звернутися в основній частині сценарію.

За замовчуванням всі змінні, визначені в сценарії, розглядаються як глобальні змінні. До змінних, певним поза функцією, можна цілком успішно звертатися безпосередньо до функцій:

**$ cat test8**

**#!/bin/bash**

**# Використання глобальної змінної для передачі значення**

**function dbl**

**{**

**value=$[$ value \* 2]**

**}**

**read -p "Enter a value:" value**

**dbl**

**echo "The new value is: $ value"**

**$ ./test8**

**Enter a value: 450**

**The new value is: 900**

Змінна **$value**визначена поза функцією, і значення їй присвоєно також за межами функції. При виконанні функції **dbl** і ця змінна, і її значення як і раніше залишаються дійсними до функцій. А якщо цій змінній буде присвоєно нове значення у функції, то таке нове значення все ще залишиться дійсним при обробці в сценарії посилання на змінну.

Але організація роботи зі змінними за таким принципом загрожує помилками, особливо якщо одна і та ж функція повинна застосовуватися в різних сценаріях командного інтерпретатора. Для цього потрібно точно знати, які змінні використовуються в функції, включаючи будь-які змінні, що застосовуються для обчислення значень, не призначених для повернення в сценарій. Нижче наведено приклад того, як може виникнути порушення в роботі.

**$ cat badtest2**

**#!/bin/bash**

**# Приклад неправильного використання змінних**

**function func1**

**{**

**temp=$[$value+5]**

**result=$[$temp\*2]**

**}**

**temp=4**

**value=6**

**func1**

**echo "The result is $ result"**

**if [ $temp -gt $value ]**

**then**

**echo "temp is larger"**

**else**

**echo "temp is smaller"**

**fi**

**$ ./badtest2**

**The result is 22**

**temp is larger**

В даному випадку змінна **$temp**застосовується і в функції, і в самому сценарії, тому відбувається зміна її значення в сценарії, не передбачене в функції, що призводить до отримання непередбаченого результату. Для вирішення цієї проблеми в функціях може застосовуватися простий спосіб, який показаний нижче.

**Локальні змінні**

Замість використання у функції глобальних змінних можна оголосити всі змінні, що застосовуються виключно в самій функції, як локальні змінні. Для цього досить задати ключове слово local перед оголошенням змінної:

**local temp**

Ключове слово **local**може бути задано також в операторі присвоєння при призначенні значення змінної:

**local temp=$[$value+5]**

Ключове слово **local**гарантує те, що область дії змінної буде обмежуватися тільки межами функції. Якщо в сценарії поза функції з'явиться змінна з тим же ім'ям, то в командному інтерпретаторі буде враховуватися, що це - дві окремі змінні з різними значеннями. Тепер можна легко забезпечити використання змінних функції окремо від змінних сценарію і надати загальний доступ тільки до тих змінним, які повинні бути доступними і в основному коді сценарію, і в функції:

**$ cat test9**

**#!/bin/bash**

**# Приклад застосування ключового слова local**

**function func1**

**{**

**local temp=$[$value+5]**

**result=$[$temp\*2]**

**}**

**temp=4**

**value=6**

**func1**

**echo "The result is $ result"**

**if [ $temp -gt $value ]**

**then**

**echo "temp is larger"**

**else**

**echo "temp is smaller"**

**fi**

**$ ./test9**

**The result is 22**

**temp is smaller**

В даному випадку зміна значення змінної **$temp**в межах функції **func1**не призводить до зміни значення, присвоєного змінній **$temp**в основному сценарії.

**Змінні типу масиву і функції**

**Передача масивів у функції**

Спосіб передачі змінної з типом масиву в функцію сценарію є достатньо складним. При спробі передати змінну з типом масиву у вигляді окремого параметра нічого не вийде:

**$ cat badtest3**

**#!/bin/bash**

**# Спроба передати змінну з типом масиву**

**function testit**

**{**

**echo "The parameters are: $@"**

**thisarray = $1**

**echo "The received array is ${thisarray[\*]}"**

**}**

**myarray=(1 2 3 4 5)**

**echo "The original array is: ${myarray[\*]}"**

**testit $myarray**

**$ ./badtest3**

**The original array is 1 2 3 4 5**

**The parameters are: 1**

**./badtest3: thisarray [\*]: bad array subscript**

**The received array is**

Якщо робиться спроба використовувати змінну з типом масиву як параметр функції, то в функцію зчитується тільки перше значення з змінної з типом масиву.

Для усунення цієї проблеми необхідно розібрати змінну з типом масиву на окремі значення, а потім задати ці значення як параметри функції. У самій же функції можна повторно зібрати всі параметри у вигляді нової змінної з типом масиву. Нижче наведено приклад виконання зазначених дій.

**$ cat test10**

**#!/bin/bash**

**# Перевірка передачі змінної з типом масиву в функцію**

**function testit**

**{**

**local newarray**

**newarray=(`echo "$@"`)**

**echo "The new array value is: ${newarray[\*]}"**

**}**

**myarray=(1 2 3 4 5)**

**echo "The original array is ${myarray[\*]}"**

**testit ${myarray[\*]}**

**$ ./test10**

**The original array is 1 2 3 4 5**

**The new array value is 1 2 3 4 5**

У цьому сценарії використовується змінна **$myarray** для збереження всіх окремих значень елементів масиву, які повинні бути поміщені в командному рядку при виклику функції. Після цього в функції знову формується змінна з типом масиву з параметрів командного рядка. Потім в цій функції отриманий масив може використовуватися точно так же, як і будь-який інший масив:

**$ cat test11**

**#!/bin/bash**

**# Складання значень в масиві**

**function addarray**

**{**

**local sum=0**

**local newarray**

**newarray=(`echo "$@"`)**

**for value in ${newarray[\*]}**

**do**

**sum=$[$sum+$value]**

**done**

**echo $sum**

**}**

**myarray=(1 2 3 4 5)**

**echo "The original array is: ${myarray[\*]}"**

**arg1=`echo ${myarray[\*]}`**

**result=`addarray $arg1`**

**echo "The result is $ result"**

**$ ./test11**

**The original array is 1 2 3 4 5**

**The result is 15**

У функції **addarray** здійснюється ітерація за значеннями елементів масиву для отримання їх загальної суми. У змінну з типом масиву **myarray** можна помістити будь-яку кількість значень, і функція **addarray** зробить їх складання.

**Повернення масивів з функцій**

При передачі змінної з типом масиву з функції назад в сценарій командного інтерпретатора може використовуватися аналогічний спосіб. У функції використовується інструкція **echo** для виведення окремих значень елементів масиву в належному порядку, а потім в сценарії повинна бути проведена повторна збірка цих елементів в якості нової змінної з типом масиву:

**$ cat test12**

**#!/bin/bash**

**# Повернення значення масиву**

**function arraydblr**

**{**

**local origarray**

**local newarray**

**local elements**

**local i**

**origarray=(‘echo "$@"`)**

**newarray=(`echo "$@"`)**

**elements=$[$#-1]**

**for ((i=0; i<=$elements; i++))**

**{**

**newarray[$i]=$[${origarray[$i]}\*2]**

**}**

**echo ${newarray[\*]}**

**}**

**myarray=(1 2 3 4 5)**

**echo "The original array is: $ {myarray [\*]}"**

**arg1=`echo ${myarray [\*]}`**

**result=(`arraydblr $arg1`)**

**echo "The new array is: ${result[\*]}"**

**$ ./test12**

**The original array is 1 2 3 4 5**

**The new array is 2 4 6 8 10**

У сценарії значення елемента масиву передається в функцію **arraydblr**за допомогою змінної **$arg1**. У функції **arraydblr** відбувається повторна збірка масиву для отримання нової змінної з типом масиву, а також формується копія для застосування в якості вихідної змінної з типом масиву. Потім в функції здійснюється ітерація по окремим значенням змінної з типом масиву, кожне значення подвоюється і поміщається в копію змінної з типом масиву.

Після цього в функції **arraydblr**використовується інструкція **echo** для виведення окремих значень змінної з типом масиву. У сценарії висновок функції **arraydblr** застосовується для повторного складання нової змінної з типом масиву на основі отриманих значень.

**Рекурсивний виклик функцій**

Однією з особливостей, якій локальні змінні наділяють функцію, є автономність. Автономна функція не використовує ніяких ресурсів за межами функції, крім тих змінних, які були передані їй сценарієм в командному рядку.

Така особливість дозволяє викликати функцію рекурсивно. Під цим мається на увазі виклик функції в самій функції для отримання відповіді. Зазвичай в рекурсивної функції передбачено базове значення, до якого в кінцевому підсумку повинна сходитися рекурсія. Рекурсія використовується в багатьох удосконалених математичних алгоритмах для поетапного зменшення ступеня складності рівняння до тих пір, поки не буде досягнутий ступінь складності, що визначається базовим значенням.

Класичним прикладом рекурсивного алгоритму є обчислення факторіала. Факторіал числа - це добуток членів натурального ряду чисел, що закінчується даними числом. Таким чином, щоб знайти факторіал числа 5, необхідно виконати наступне обчислення:

**5! = 1 \* 2 \* 3 \* 4 \* 5 = 120**

З використанням рекурсії наведене тут вираз можна скорочено уявити в такій формі:

**x! = X \* (x-1)!**

або, в описовому вигляді: "факторіал x дорівнює добутку x на факторіал x-1". Таке обчислення може бути реалізовано за допомогою наступного простого рекурсивного сценарію:

**function factorial**

**{**

**if [$ 1 -eq 1]**

**then**

**echo 1**

**else**

**local temp = $ [$ 1 - 1]**

**local result = 'factorial $ temp'**

**echo $ [$ result \* $ 1]**

**fi**

**}**

Функція **factorial**викликає саму себе для обчислення значення факторіала.

**Створення бібліотеки**

Як вже було сказано, функції дозволяють позбутися від необхідності багаторазово вводити однаковий код в одному сценарії, але іноді повтори коду зустрічаються в різних сценаріях. Очевидно, що в такому випадку немає сенсу визначати в кожному сценарії функцію, що включає повторюється код, лише для того, щоб використовувати її по одному разу в кожному сценарії.

Для цієї проблеми передбачено ефективне рішення. Командний інтерпретатор **bash**дозволяє створити файл бібліотеки для власних функцій, а потім замість повторного введення визначень функцій посилатися на цей файл бібліотеки у всіх сценаріях, де будуть потрібні представлені в ньому функції.

Перший крок в цьому процесі полягає в створенні загального файлу бібліотеки, що містить функції, які можуть знадобитися в сценаріях. Нижче наведено приклад простого файлу бібліотеки з ім'ям **myfuncs**, в якому визначені три нескладні функції:

**$ cat myfuncs**

**# Функції для застосування в сценаріях**

**function addem**

**{**

**echo $[$1+$2]**

**}**

**function multem**

**{**

**echo $[$1\*$2]**

**}**

**function divem**

**{**

**if [ $2 -ne 0 ]**

**then**

**echo $[$1/$2]**

**else**

**echo -1**

**fi**

**}**

Команда **source**має псевдонім, застосовуваний для її скороченого позначення, який прийнято називати оператором точки. Щоб використовувати в сценарії командного інтерпретатора в якості джерела (**source**) текст з файлу бібліотеки **myfuncs**, достатньо ввести в сценарій наступний рядок:

**.** **./myfuncs**

В даному прикладі передбачається, що файл бібліотеки **myfuncs** знаходиться в тому ж каталозі, що і сценарій командного інтерпретатора. В іншому випадку буде потрібно задати відповідний шлях доступу до цього файлу. Нижче наведено приклад створення сценарію, в якому використовується файл бібліотеки **myfuncs**.

**$ cat test14**

**#!/bin/bash**

**# Використання функцій, визначених у файлі бібліотеки**

**.** **./myfuncs**

**value1=10**

**value2=5**

**result1=`addem $value1 $value2`**

**result2=`multem $value1 $value2`**

**result3=`divem $value1 $value2`**

**echo "The result of adding them is: $result1"**

**echo "The result of multiplying them is: $result2"**

**echo "The result of dividing them is: $result3"**

**$ ./test14**

**The result of adding them is: 15**

**The result of multiplying them is: 50**

**The result of dividing them is: 2**

У цьому сценарії виклик функцій, визначених у файлі бібліотеки **myfuncs**, відбувається цілком успішно.

**Лекція №8**

**«Система X Window для роботи з графічними середовищами у ОС Linux. Сценарії bash для графічних робочих столів»**

**Графічні робочі столи Linux**

Конфігурація робочого столу Linux є дуже гнучкою. Велика частина того, з чим доводиться мати справу користувачеві Linux («враження і відчуття» від використання робочого стола), виходить від додатків або їх будівельних блоків. Якщо вам не подобається який-небудь додаток, то, як правило, можна знайти йому заміну. Якщо те, що ви шукаєте, поки ще не існує, ви можете створити це самостійно. Розробники Linux прагнуть до того, щоб у робочого столу була велика різноманітність можливих варіантів настройки, це призводить до більшого вибору.

Для спільної роботи всіх програм необхідно мати щось спільне, і в серцевині майже всього в більшості робочих столів Linux знаходиться **X-сервер (сервер системи X Window)** . Розглядайте його як свого роду «ядро» робочого столу, яке керує всім, починаючи з рендеринга вікон і конфігурації дисплеїв і завершуючи обробкою введення від таких пристроїв, як клавіатура і миша. X-сервер є також тим компонентом, яким ви насилу зможете підшукати заміну.

**X-сервер**- це всього лише сервер, і він не нав'язує спосіб дії або відображення чогось. Замість нього з призначеним для користувача інтерфейсом працюють команди **X-клієнта** . Базові програми **X-клієнта**, наприклад вікна терміналу і браузери, підключаються до **X-сервера** і просять його намалювати вікна. У відповідь на це **X-сервер** з'ясовує, де розмістити ці вікна, і виконує рендеринг. Там, де необхідно, **X-сервер** відправляє також введення назад клієнту.

**X-клієнти**не повинні діяти подібно додаткам, реалізованим за допомогою вікон; вони можуть діяти як служби для інших клієнтів або забезпечувати інші функції інтерфейсу. Менеджер вікна є, ймовірно, найважливішим додатком служби клієнта, оскільки він обчислює, як організувати вікна на екрані, і постачає їх інтерактивними «прикрасами» на кшталт заголовної рядки, яка дозволяє користувачеві переміщати і мінімізувати вікна. Ці можливості є центральними для роботи користувача.

Незважаючи на те що інструментарій забезпечує уніфіковане оформлення, для деяких деталей робочого столу потрібний певний ступінь кооперації між різними додатками. Наприклад, одному з додатків може знадобитися спільно використовувати дані з іншим додатком або оновити загальну панель повідомлень на робочому столі. Щоб задовольнити ці потреби, інструментарії та інші бібліотеки об'єднуються в великі пакети, звані оточеннями робочого столу. Поширеними оточеннями робочого столу Linux є **GNOME**, **KDE**, **Unity** і **Xfce** .

**Система X Window**

Система **X Window** (http://www.x.org/) історично склалася дуже великою, з основним дистрибутивом, що включає **X-сервер**, бібліотеки підтримки клієнтів і самих клієнтів. Внаслідок появи таких середовищ робочого столу, як **GNOME** та **KD** , роль пакета **X** з часом змінилася, і тепер акцент зроблений на основному сервері, який управляє рендерингом і пристроями введення, а також на спрощення бібліотеки клієнтів.

**X-сервер**легко виявити в системі. Він називається **X**. Його можна знайти в списку процесів; зазвичай виявляється, що він запущений з деякою кількістю параметрів, наприклад, так:

**/usr/bin/X :0 -auth /var/run/lightdm/root/ :0 -nolisten tcp vt7 -novtswitch**

Параметр **:0**, показаний тут, називається дисплеєм. Це ідентифікатор, який представляє один або кілька моніторів, до яких ми отримуємо доступ за допомогою клавіатури і/або миші. Зазвичай дисплей відповідає єдиному монітору, який підключений до нашого комп'ютера, але ми можемо помістити кілька моніторів за одним і тим же дисплеєм. При використанні **X-сеансу** для змінної оточення **DISPLAY** встановлено значення ідентифікатора дисплея.

У Linux **X-сервер** запускається у віртуальному терміналі. В даному прикладі аргумент **vt7** говорить нам про те, що сервер був запущений в терміналі **/dev/tty7**(зазвичай сервер запускається в першому доступному віртуальному терміналі). Можна запустити більше одного **X-сервера** в даний момент часу, надавши кожному з них окремий віртуальний термінал, але в такому випадку для кожного сервера потрібно унікальний ідентифікатор дисплея. Між серверами можна перемикатися за допомогою поєднання клавіш **Ctrl+Alt+Fn**або команди **chvt**.

Зазвичай **X-сервер** не запускають за допомогою командного рядка, оскільки запуск цього сервера не визначає ніяких клієнтів, призначених для роботи з ним. Якщо ви запустите сервер сам по собі, ви просто отримаєте порожній екран. Замість цього найпоширенішим способом запуску **X-сервера** є використання менеджера дисплея - утиліти, яка запускає сервер і поміщає на екран вікно входу в систему. Коли ви виконаєте вхід, менеджер дисплея запускає ряд клієнтів, таких як менеджер вікон і менеджер файлів, щоб ви змогли почати використання комп'ютера.

Існує багато різних менеджерів дисплея, наприклад **gdm** (для середовища **GNOME**) і **kdm** (для середовища **KDE**). Аргумент **lightdm**, присутній в наведеному вище виклику X-сервера, є крос-платформним менеджером дисплея, призначеним для запуску в сеансах **GNOME** або **KDE**.

Щоб запустити X-сеанс з віртуальної консолі, а не за допомогою менеджера дисплея, можна виконати команду **startx**або **xinit**. Однак сесія, яку ви в результаті отримаєте, буде, ймовірно, дуже простою і абсолютно відрізняється від тієї, яка створюється менеджером дисплея, оскільки їх функції та стартові файли різні.

**Аналоги пакету dialog для графічних робочих столів KDE та GNOME**

Додаткову привабливість інтерактивним сценаріями може надати більш широке застосування графічних засобів, для чого необхідно пройти на крок далі. У варіантах середовища робочого столу **KDE** і **GNOME** задум команди **dialog**був в ще більшому ступені вдосконалений і включені команди, за допомогою яких формуються графічні елементи X Window для відповідного середовища.

**Середовище KDE**

Графічне середовище **KDE** включає за замовчуванням пакет **kdialog**. У пакеті **kdialog**використовується команда **kdialog**для формування в додатках для робочого столу **KDE** стандартних вікон, аналогічних графічним елементам в стилі команди **dialog**. Але ці вікна мають таку особливість, що не виглядають як інородне тіло, а бездоганно поєднуються з іншими вікнами додатка **KDE**. Це дозволяє створювати призначені для користувача інтерфейси, які не поступаються за якістю інтерфейсів додатків Windows, безпосередньо в сценаріях командного інтерпретатора.

Однак слід пам'ятати, що застосування в дистрибутиві Linux робочого столу **KDE** не означає, що пакет **kdialog**обов'язково встановлюється за умовчанням. Може знадобитися встановити цей пакет вручну, отримавши його з сховища конкретного дистрибутива.

За аналогією з командою **dialog**, команда **kdialog**передбачає завдання опцій командного рядка для вказівки на те, який графічний елемент вікна повинен використовуватися.

Команда **kdialog**має такий вигляд:

**kdialog display-options window-options arguments**

Опції **window-options**дозволяють визначати призначений для використання графічний елемент вікна. Опції, передбачені в цій команді, наведені в табл. 12.2.

Табл. 12.2. Опції вікна **kdialog**

|  |  |
| --- | --- |
| **параметр** | **опис** |
| **--checklist title [tag item status]** | Компонент меню у вигляді списку вибору, в якому параметр **status**вказує, відзначений елемент списку чи ні |
| **--error text** | Поле повідомлення про помилку |
| **--inputbox text [init]** | Текстове поле введення. Передбачена можливість задати текстовий рядок, що відображається в полі **text**за замовчуванням, за допомогою значення **init** |
| **--menu title [tag item]** | Заголовок вікна меню у вигляді списку вибору і список елементів, позначених дескрипторами |
| **--msgbox text** | Просте вікно повідомлення із вказаним текстом |
| **--password text** | Текстове поле введення пароля, яке приховує введення даних користувачем |
| **--radiolist title [tag item status]** | Меню у вигляді списку перемикачів, де параметр status вказує, обраний елемент чи ні |
| **--separate-output** | Команда повертає елементи меню зі списками вибору і списками перемикачів в окремих рядках |
| **--sorry text** | Вікно повідомлення "sorry" (шкодуємо) |
| **--textbox file [width] [height]** | Текстове поле, яке відображає вміст файлу **file**, для якого додатково можна вказати ширину і висоту, **width**і **height** |
| **-title title** | Задає назву для області **TitleBar** діалогового вікна |
| **--warningyesno text** | Вікно з попереджуючим повідомленням, в якому передбачені кнопки **Yes (Так)** і **No (Ні)** |
| **--warningcontinuecancel text** | Вікно з попереджуючим повідомленням, в якому передбачені кнопки **Continue (Продовжити)** і **Cancel (Скасування)** |
| **--warningyesnocancel text** | Вікно з попереджуючим повідомленням, в якому передбачені кнопки **Yes (Так)** , **No (Ні)** , **Cancel (Скасування)** |
| **--yesno text** | Поле з питанням, в якому передбачені кнопки **Yes (Так)** і **No (Ні)** |
| **--yesnocancel text** | Поле з питанням, в якому передбачені **Yes (Так)** , **No (Ні)** , **Cancel (Скасування)** |

Графічні елементи вікна **kdialog**можна використовувати в сценаріях командного інтерпретатора за аналогією з тим, як використовуються графічні елементи **dialog**. Істотною відмінністю є те, що графічні елементи вікна kdialog передають вихідні значення в потік **STDOUT**, а не в потік **STDERR** .

Нижче наведено сценарій, який дозволяє перетворити наведене вище меню системного адміністратора в додаток **KDE** .

**$ cat menu4**

**#!/bin/bash**

**# Використання команди kdialog для створення меню**

**temp=`mktemp -t temp.XXXXXX`**

**temp2=`mktemp -t temp2.XXXXXX`**

**function diskspace**

**{**

**df -k> $ temp**

**kdialog --textbox $temp 1000 10**

**}**

**function whoseon**

**{**

**who > $temp**

**kdialog --textbox $temp 500 10**

**}**

**function memusage**

**{**

**cat /proc/meminfo > $temp**

**kdialog --textbox $temp 300 500**

**}**

**while [1]**

**do**

**kdialog --menu "Sys Admin Menu" "1" "Display diskspace" "2" "Display**

**users "" 3 "" Display memory usage "" 0 "" Exit "> $ temp2**

**if [ $? -eq 1 ]**

**then**

**break**

**fi**

**selection=`cat $temp2`**

**case $selection in**

**1)**

**diskspace ;;**

**2)**

**whoseon ;;**

**3)**

**memusage ;;**

**0)**

**break ;;**

**\*)**

**kdialog --msgbox "Sorry, invalid selection"**

**esac**

**done**

**Середовище GNOME**

Графічне середовище **GNOME** підтримує наступні два популярних пакету, які дозволяють формувати стандартні вікна:

* **gdialog;**
* **zenity.**

У порівнянні з **gdialog**пакет **zenity**є набагато більш поширеним і передбачений в більшості дистрибутивів Linux з робочим столом **GNOME** (зокрема, він встановлюється за умовчанням в дистрибутивах Ubuntu і Fedora).

Команда **zenity**дозволяє створювати різні графічні елементи вікон з використанням опцій командного рядка. У табл. 12.3 перераховані різні графічні елементи, які можуть бути сформовані за допомогою команди **zenity**.

Табл. 12.3. Графічні елементи вікон, що формуються за допомогою команди zenity

|  |  |
| --- | --- |
| **параметр** | **опис** |
| ***--calendar*** | Відобразити повний календар на місяць |
| ***--entry*** | Показати діалогове вікно введення тексту |
| ***--error*** | Показати діалогове вікно повідомлення про помилку |
| ***--file-selection*** | Відобразити діалогове вікно з повним ім'ям шляху і ім'ям файлу |
| ***--info*** | Відобразити інформаційне діалогове вікно |
| ***--list*** | Вивести діалогове вікно зі списком вибору або списком перемикачів |
| ***--notification*** | Відобразити значок повідомлення |
| ***--progress*** | Вивести діалогове вікно з індикатором ходу роботи |
| ***--question*** | Вивести діалогове вікно з питанням і кнопками **Yes (Так)** і **No (Ні)** |
| ***--scale*** | Відобразити діалогове вікно визначення масштабу |
| ***--text-info*** | Відобразити текстове поле, що містить текст |
| ***--warning*** | Відобразити діалогове вікно з попередженням |

Програми для командного рядка з командою **zenity**діють трохи інакше в порівнянні з програмами на основі **kdialog**і **dialog**. Зокрема, багато типів графічних елементів визначаються з використанням додаткових опцій в командному рядку, а не задаються шляхом включення їх параметрів опцій.

Команда **zenity**дуже добре проявляє себе в сценаріях командного інтерпретатора. Але, на жаль, розробники пакету **zenity**відмовилися від дотримання угод по вибору опцій, які використовуються в пакетах **dialog**і **kdialog**, тому при перетворенні будь-яких існуючих інтерактивних сценаріїв в сценарії для команди **zenity**можуть виникати складнощі.

Зокрема, при реалізації меню системного адміністратора з переходом від **kdialog**до **zenity**з визначеннями графічних елементів доводиться виконувати досить багато маніпуляцій.

**$ cat menu5**

**#!/bin/bash**

**# Використання команди zenity для створення меню**

**temp = 'mktemp -t temp.XXXXXX'**

**temp2 = 'mktemp -t temp2.XXXXXX'**

**function diskspace**

**{**

**df –k > $temp**

**zenity --text-info --title "Disk space" --filename=$temp --width 750 --height 10**

**}**

**function whoseon**

**{**

**who > $temp**

**zenity --text-info --title "Logged in users" - filename = $ temp --width 500 --height 10**

**}**

**function memusage**

**{**

**cat /proc/meminfo > $temp**

**zenity --text-info --title "Memory usage" --filename = $ temp --width 300 --height 500**

**}**

**while [ 1 ]**

**do**

**zenity --list --radiolist --title "Sys Admin Menu" --column "Select" --column "Menu Item" FALSE "Display diskspace" FALSE "Display users" FALSE "Display memory usage" FALSE "Exit"> $ temp2**

**if [ $? -eq 1 ]**

**then**

**break**

**fi**

**selection=`cat $temp2`**

**case $selection in**

**"Display disk space")**

**diskspace ;;**

**"Display users")**

**whoseon ;;**

**"Display memory usage")**

**memusage ;;**

**Exit)**

**break ;;**

**\*)**

**zenity --info "Sorry, invalid selection"**

**esac**

**done**

Команда **zenity**не підтримує діалогове вікно меню, тому для головного меню довелося використовувати вікно меню, що діє за принципом списку перемикачів.

За допомогою пакета **zenity**можна домогтися того, що пропозиції для робочого столу **GNOME** , сформовані за допомогою інтерактивних сценаріїв командного інтерпретатора, будуть виглядати як додатки Windows.

**Лекція №9**

**«Команди для роботи з мережею у Linux»**

Коли справа доходить до мережевих можливостей, важко уявити щось, що було б неможливо для Linux. Linux використовується для створення всіх видів мережевих систем, програмних компонентів і пристроїв, включаючи брандмауери, маршрутизатори, сервери імен, мережеві пристрої зберігання даних (Network-Attached Storage, NAS) і так далі і тому подібне.

Наскільки велика тема мереж, настільки ж велика колекція команд, які можна використовувати для налаштування і управління ними. Ми зосередимо свою увагу лише на тих з них, які найчастіше використовуються на практиці. У число команд, обраних для дослідження в цій главі, входять команди, використовувані для моніторингу мереж і передачі файлів. Додатково ми досліджуємо програму ***ssh***, використовувану для входу в віддалені системи. Ми розглянемо наступні команди:

* ***ping*** - посилає пакети ICMP ECHO\_REQUEST вузлів в мережі;
* ***traceroute*** - виводить трасування маршруту пакетів до мережевого вузла;
* ***netstat*** - виводить список мережевих з'єднань, таблиці маршрутів, статистику інтерфейсів, масковані з'єднання і відомості про членство в широкомовних групах;
* ***ftp*** - програма передачі файлів через Інтернет;
* ***lftp*** - поліпшена програма передачі файлів через Інтернет;
* ***wget*** - неінтерактивний завантажувач файлів з мережі;
* ***ssh*** - клієнт OpenSSH SSH (програма для входу в віддалені системи);
* ***scp*** - програма безпечного копіювання файлів через мережу;
* ***sftp*** - програма безпечної передачі файлів через мережу.

***Дослідження та моніторинг мережі***

Навіть якщо ви не є системним адміністратором, буває корисно вміти оцінювати продуктивність і функціонування мережі.

***ping - передача спеціальних пакетів мережевим вузлам***

Команда ***ping*** є найпростішою мережевою командою. Вона посилає спеціальні мережеві пакети IMCP ECHO\_REQUEST вказаному мережевому вузлу. Більшість мережевих пристроїв приймає ці пакети і відповідає на них, - це дозволяє перевірити мережеві з'єднання.

Наприклад, за допомогою команди ***ping*** можна перевірити досяжність мережного вузла **http://www.linuxcommand.org/**:

***$ ping linuxcommand.org***

Відразу після запуску програма ***ping*** починає посилати пакети з певним інтервалом (за замовчуванням 1 секунда), поки її виконання не буде перервано:

***$ ping linuxcommand.org***

***PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.***

***64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=1 ttl=43 time=10 7 ms***

***64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=2 ttl=43 time=10 8 ms***

***64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=3 ttl=43 time=10 6 ms***

***64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=4 ttl=43 time=10 6 ms***

***64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=5 ttl=43 time=10 5 ms***

***64 bytes from vhost.sourceforge.net (66.35.250.210): icmp\_seq=6 ttl=43 time=10 7 ms***

***--- linuxcommand.org ping statistics ---***

***6 packets transmitted, 6 received, 0% packet loss, time 6010ms***

***rtt min/avg/max/mdev = 105.647/107.052/108.118/0.824 ms***

Після переривання натисканням ***<CTRL>+<C>*** (в даному прикладі після шостого пакету) ***ping*** виводить результати своєї роботи. Якщо мережа функціонує належним чином, число втрачених пакетів (packet loss) становитиме нуль відсотків. Успішна робота ***ping*** може служити ознакою того, що мережеві компоненти (інтерфейсні карти, кабелі, маршрутизатори і шлюзи) знаходяться в робочому стані.

***traceroute - трасування шляху мережевих пакетів***

Програма ***traceroute*** (в деяких системах використовується схожа на неї програма ***tracepath***) виводить список всіх «переходів» (hops) на шляху мережевого трафіку між локальною системою і зазначеним вузлом мережі. Наприклад, побачити, як виглядає маршрут до сайту **http://www.slashdot.org/**, можна за допомогою наступної команди:

***$ traceroute slashdot.org***

Її виведення виглядає приблизно так:

***traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte packets***

***1 ipcop.localdomain (192.168.1.1) 1.066 ms 1.366 ms 1.720 ms***

***2 \* \* \****

***3 ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9) 14.622 ms 14.885 ms 15.169 ms***

***4 po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154) 17.634 ms 17.626 ms 17.899 ms***

***5 po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158) 15.992 ms 15.983 ms 16.256 ms***

***6 po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5) 22.835 ms 14.23 3 ms 14.405 ms***

***7 po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34) 16.154 ms 13.600 ms 18.867 ms***

***8 te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77) 21.951 ms 21.073 ms 21.557 ms***

***9 pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10) 22.917 ms 21 .884 ms 22.126 ms***

***10 204.70.144.1 (204.70.144.1) 43.110 ms 21.248 ms 21.264 ms***

***11 cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 21.857 ms cr2-pos-0-0-3-1.newyork.savvis.net (204.70.204.238) 19.556 ms cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 19.634 ms***

***12 cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109) 41.586 ms 42.843 ms cr2-tengig-0-0-2-0.chicago.savvis.net (204.70.196.242) 43.115 ms***

***13 hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net (204.70.195.122) 44.215 ms 41.833 ms 45.658 ms***

***14 csr1-ve241.elkgrovech3.savvis.net (216.64.194.42) 46.840 ms 43.372 ms 47.041 ms***

***15 64.27.160.194 (64.27.160.194) 56.137 ms 55.887 ms 52.810 ms***

***16 slashdot.org (216.34.181.45) 42.727 ms 42.016 ms 41.437 ms***

Тут можна бачити, що на шляху між нашою тестовою системою і **http://www.slashdot.org/** знаходиться 16 маршрутизаторів. Для маршрутизаторів, що надають таку нормативну інформацію, виводяться імена хостів, IP-адреси і інформація про продуктивність, яка включає три інтервалу часу, що знадобилися для передачі / підтвердження пакетів між локальною системою і маршрутизатором. Для маршрутизаторів, що не надають ідентифікаційної інформації (наприклад, через особливості настройки маршрутизатора, заторів у мережі, дій брандмауерів і т. д.), Виводяться зірочки, як це можна бачити в рядку, що відповідає другому переходу.

***netstat*** ***- виведення параметрів настройки мережі і статистик***

Програма ***netstat*** використовується для дослідження різних налаштувань мережі та статистик. За допомогою множини параметрів цієї команди можна переглядати різні аспекти настройки мережі. За допомогою параметра ***-ie***, наприклад, можна досліджувати мережеві інтерфейси в системі:

***$ netstat -ie***

***eth0 Link encap:Ethernet HWaddr 00:1d:09:9b:99:67***

***inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0***

***inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link***

***UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1***

***RX packets:238488 errors:0 dropped:0 overruns:0 frame:0***

***TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0***

***collisions:0 txqueuelen:100***

***RX bytes:153098921 (146.0 MB) TX bytes:261035246 (248.9 MB)***

***Memory:fdfc0000-fdfe0000***

***lo Link encap:Local Loopback***

***inet addr:127.0.0.1 Mask:255.0.0.0***

***inet6 addr: ::1/128 Scope:Host***

***UP LOOPBACK RUNNING MTU:16436 Metric:1***

***RX packets:2208 errors:0 dropped:0 overruns:0 frame:0***

***TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0***

***collisions:0 txqueuelen:0***

***RX bytes:111490 (108.8 KB) TX bytes:111490 (108.8 KB)***

Приклад, наведений вище, показує, що наша тестова система має два мережевих інтерфейси. Перший, з ім'ям **eth0**, - це інтерфейс Ethernet; другий, з ім'ям **lo**, - це петлевий інтерфейс (loopback), віртуальний інтерфейс, який система використовує, щоб розмовляти «сама з собою».

Виконуючи причинно-наслідковий діагностику, перше, на що слід звернути увагу, - наявність слова UP на початку четвертого рядка для кожного інтерфейсу, в якому зазначається, що мережевий інтерфейс включений, і присутність допустимого IP-адреси в поле **inet addr** у другому рядку. Для систем, що використовують протокол динамічної настройки хостів (Dynamic Host Configuration Protocol, DHCP), наявність допустимого IP-адреси в цьому полі підтвердить нормальну роботу DHCP.

Використання параметра ***-r*** дозволить отримати таблицю маршрутизації ядра. З цієї таблиці можна судити, як налаштована передача пакетів між мережами:

***$ netstat -r***

***Таблиця маршрутизації ядра протоколу IP***

***Destination Gateway Genmask Flags MSS Window irtt Iface***

***192.168.1.0 \* 255.255.255.0 U 0 0 0 eth0***

***default 192.168.1.1 0.0.0.0 UG 0 0 0 eth0***

У цьому простому прикладі представлена ​​типова таблиця маршрутизації для клієнтської машини, підключеної до локальної мережі (Local Area Network, LAN), що знаходиться за брандмауером / маршрутизатором. У першому рядку демонструється адреса призначення **192.168.1.0**. IP-адреси, що закінчуються нулем, відповідають цілим мережам, а не окремим вузлам в них, тому таку адресу має на увазі: «будь-який вузол в локальній мережі». Наступне поле, **Gateway** (шлюз), визначає ім'я або IP-адресу шлюзу (маршрутизатора) для виходу поточного вузла в зазначену мережу. Зірочка в цьому полі вказує, що використовувати шлюз не потрібно.

В останньому рядку в якості адреси призначення вказано слово **default** (за замовчуванням). Цей рядок управляє трафіком, адресованим будь-яких мереж, що не наведені в таблиці. В даному прикладі видно, що роль шлюзу виконує маршрутизатор з адресою **192.168.1.1**, який, по всій видимості, знає, що робити з трафіком.

Програма ***netstat*** має безліч параметрів, з яких ми розглянули лише пару. Повний їх список ви знайдете на сторінці довідкового керівництва (***man***) для ***netstat***.

***Передача файлів по мережі***

Що толку від мережі, якщо не знати, як переміщати файли через неї? Існує безліч програм, які переміщують дані по мережі. Ми розглянемо дві з них.

***ftp - передача файлів по протоколу FTP***

Одна з по-справжньому «класичних» програм - ***ftp*** - отримала своє ім'я від використовуваного нею протоколу, протоколу передачі файлів (File Transfer Protocol, FTP). Протокол **FTP** широко використовується в Інтернеті для передачі файлів. Він підтримується більшістю веб-браузерів, якщо не всіма, і вам часто будуть зустрічатися ідентифікатори URI, що починаються із префікса протоколу ftp://.

Програма ***ftp*** з'явилася задовго до веб-браузерів. Вона використовувалася для обміну даними з серверами **FTP**, комп'ютерами, що зберігають файли, які можна вивантажувати і завантажувати по мережі.

Протокол **FTP** (в своєму первісному вигляді) небезпечний, тому що пересилає імена і паролі у відкритому текстовому вигляді. Тобто вони не шифруються, і будь-хто, хто здатний перехопити мережевий трафік, зможе побачити їх. З цієї причини практично всі операції по протоколу **FTP** в Інтернеті виконуються анонімними серверами **FTP**. Анонімний сервер дозволяє будь-якому охочому підключитися з обліковим записом **anonymous** без пароля.

У наступному прикладі показаний типовий сеанс роботи з програмою ***ftp*** для завантаження ISO-образу з дистрибутивом Ubuntu з каталогу ***/pub/cd\_images/Ubuntu-8.04*** анонімного сервера **FTP fileserver**.

***$ ftp fileserver***

***Connected to fileserver.localdomain.***

***220 (vsFTPd 2.0.1)***

***Name (fileserver:me): anonymous***

***331 Please specify the password.***

***Password:***

***230 Login successful.***

***Remote system type is UNIX.***

***Using binary mode to transfer files.***

***ftp> cd pub/cd\_images/Ubuntu-8.04***

***250 Directory successfully changed.***

***ftp> ls***

***200 PORT command successful. Consider using PASV.***

***150 Here comes the directory listing.***

***-rw-rw-r-- 1 500 500 733079552 Apr 25 03:53 ubuntu-8.04-desktopi386.iso***

***226 Directory send OK.***

***ftp> lcd Desktop***

***Local directory now /home/me/Desktop***

***ftp> get ubuntu-8.04-desktop-i386.iso***

***local: ubuntu-8.04-desktop-i386.iso remote: ubuntu-8.04-desktop-i386.iso***

***200 PORT command successful. Consider using PASV.***

***150 Opening BINARY mode data connection for ubuntu-8.04-desktop-i386.iso***

***(733079552 bytes).***

***226 File send OK.***

***733079552 bytes received in 68.56 secs (10441.5 kB/s)***

***ftp> bye***

У табл. 9.1 наводиться опис команд, що вводилися в ході цього сеансу.

Табл. 9.1. Приклади команд інтерактивного сеансу ***ftp***

|  |  |
| --- | --- |
| **Команда** | **Опис** |
| ***ftp fileserver*** | Викликає програму ***ftp*** і пропонує їй з'єднатися з сервером ***fileserver*** |
| ***anonymous*** | Ім'я користувача для входу. Після введення імені користувача з'явиться запрошення ввести пароль. Деякі сервери приймають порожній пароль. Інші можуть вимагати пароль в форматі адреси електронної пошти. В даному випадку спробуйте що-небудь схоже на ***user@example.com*** |
| ***cd pub/cd\_images/Ubuntu-8.04*** | Виконує перехід в каталог у віддаленій системі, де знаходиться потрібний файл. Зверніть увагу, що на більшості анонімних серверів FTP файли, доступні для завантаження всім бажаючим, знаходяться десь в каталозі ***pub*** |
| ***ls*** | Виводить вміст каталогу в віддаленій системі |
| ***lcd Desktop*** | Виконує перехід в каталог ***~/Desktop*** в локальній системі. У цьому прикладі програма ***ftp*** була викликана в поточному робочому каталозі ***~***. Дана команда призначає поточним робочим каталог ***~/Desktop*** |
| ***get ubuntu-8.04-desktop-i386.iso*** | Посилає віддаленій системі запит на передачу файлу **ubuntu-8.04-desktop-i386.iso** локальній системі. Так як в локальній системі поточним робочим обраний каталог ***~/ Desktop***, файл буде завантажений в нього |
| ***bye*** | Виходить з віддаленої системи і завершує сеанс програми ***ftp***. Також можна використовувати команди ***quit*** і ***exit*** |

Якщо в запрошенні ***ftp>*** ввести команду ***help***, програма виведе список підтримуваних команд. За допомогою програми ***ftp*** можна виконувати безліч звичайних операцій з файлами на сервері, правда, при наявності достатніх привілеїв. Це не дуже зручно, але здійснимо.

***lftp - більш вдала версія ftp***

***ftp*** - не єдиний клієнт FTP командного рядка. Насправді таких клієнтів безліч. Одним з кращих (і більш популярним) вважається ***lftp*** Олександра Лук'янова. Цей клієнт діє майже так само, як традиційна програма ***ftp***, але має безліч додаткових функцій, включаючи підтримку декількох протоколів (в тому числі і HTTP), можливість автоматичного відновлення перервалася завантаження, виконання операцій у фоновому режимі, автодоповнення шляхів по клавіші Tab і багато іншого.

***wget - неінтерактивний завантажувач файлів з мережі***

***wget*** - ще одна популярна програма командного рядка для завантаження файлів. Її зручно використовувати для завантаження вмісту веб-і FTP-сайтів. За допомогою ***wget*** можна завантажити один файл, кілька файлів і навіть цілий сайт. Наприклад, завантажити першу сторінку сайту **http://www.linuxcommand.org/** можна командою:

***$ wget http://linuxcommand.org/index.php***

***--11:02:51-- http://linuxcommand.org/index.php***

***=> `index.php'***

***Розпізнається linuxcommand.org... 66.35.250.210***

***Підключення до linuxcommand.org|66.35.250.210|:80... з’єднання встановлено.***

***HTTP-запит відправлено. Очікування відповіді… 200 OK***

***Довжина: 3808 (3.7K) [text/html]***

***Збереження в: "index.php"***

***[ <=> ] 3,120 --.--KB/s***

***11:02:51 (161.75 MB/s) - `index.php' збережений [3808/30808]***

Більшість параметрів, які підтримує програма ***wget***, дозволяє організувати рекурсивне отримання, завантаження файлів у фоновому режимі (дозволяє вийти з системи без зупинки завантаження) і довантажувати частково завантажені файли. Ці можливості добре описані на сторінці довідкового керівництва (***man***).

***Безпечні взаємодії з віддаленими вузлами***

Уже багато років UNIX-подібні операційні системи підтримують можливість віддаленого адміністрування по мережі. На першому етапі, ще до повсюдного поширення Інтернету, існувала пара популярних програм для входу в віддалені мережеві вузли: ***rlogin*** і ***telnet***. Однак ці програми страждали тим же фатальним недоліком, що і програма ***ftp***; всі дані (включаючи імена користувачів і паролі) вони передавали у вигляді відкритого тексту. Це абсолютно неприпустимо в епоху Інтернету.

***ssh - безпечний вхід в віддалені комп'ютери***

Для вирішення описаної проблеми був розроблений протокол з назвою **SSH** (Secure Shell - безпечна командна оболонка). **SSH** вирішує дві основні проблеми безпечної взаємодії з віддаленими мережними вузлами. По-перше, він підтверджує, що віддалений вузол є саме тим, за кого себе видає (це запобігає атаки типу «зловмисник в середині» (man-in-the-middle), і, по-друге, шифрує всі дані, що передаються між локальним і віддаленим вузлами.

У своїй роботі протокол **SSH** спирається на два компонента. На віддаленому вузлі діє сервер **SSH**, який приймає з'єднання на порту 22, а в локальній системі діє клієнт **SSH**, який здійснює обмін інформацією з віддаленим сервером.

Більшість дистрибутивів Linux включають реалізацію **SSH** з назвою **OpenSSH** з проекту **BSD**. Деякі дистрибутиви (наприклад, Red Hat) за замовчуванням містять пакети з обома компонентами, сервером і клієнтом, тоді як інші (наприклад, Ubuntu) включають тільки клієнта. Щоб система могла приймати віддалені з'єднання, в ній повинен бути встановлений пакет з реалізацією сервера OpenSSH-server, цей сервер повинен бути налаштований і запущений, і якщо система знаходиться за брандмауером, останній повинен пропускати вхідні з'єднання на порт TCP з номером 22.

Програма клієнта **SSH**, яка використовується для підключення до серверів **SSH**, має досить очевидне ім'я: ***ssh***. Підключитися до віддаленого мережевого вузла з ім'ям **remote-sys** можна за допомогою програми клієнта ***ssh***, як показано нижче:

***$ ssh remote-sys***

***The authenticity of host 'remote-sys (192.168.1.4)' can't be established.***

***RSA key fingerprint is 41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.***

***Are you sure you want to continue connecting (yes/no)?***

При першій спробі підключення на екран виводиться попередження, що повідомляє, що автентичність віддаленого вузла не може бути встановлена. Це пояснюється тим, що програма-клієнт раніше ніколи не підключалася до даного віддаленого вузла. Щоб прийняти ідентифікаційні дані віддаленого вузла, введіть ***yes*** у відповідь на запрошення. Після установки з'єднання користувачеві буде запропоновано ввести пароль:

***Warning: Permanently added 'remote-sys, 192.168.1.4' (RSA) to the list of known hosts.***

***me@remote-sys's password:***

Після введення дійсного пароля в терміналі з'явиться запрошення командної оболонки з віддаленої системи:

***Last login: Tue Aug 30 13:00:48 2011***

***[me@remote-sys ~]$***

Сеанс з віддаленою командною оболонкою триває, поки користувач не введе команду ***exit*** в запрошенні віддаленої командної оболонки, після чого з'єднання закриється. У цей момент відновиться сеанс локальної командної оболонки і з'явиться її запрошення до введення.

До віддаленої системи можна також підключитися з іншим ім'ям користувача. Наприклад, якщо локальний користувач **me** має в віддаленій системі обліковий запис з ім'ям **bob**, він зможе увійти в віддалену систему з ім'ям **bob**, виконавши наступну команду:

***$ ssh bob@remote-sys***

***bob@remote-sys's password:***

***Last login: Tue Aug 30 13:03:21 2011***

***[bob@remote-sys ~]$***

Як зазначалося вище, ***ssh*** перевіряє автентичність віддаленого вузла. Якщо віддалений вузол не пройде аутентифікацію, з'явиться наступне попередження:

***$ ssh remote-sys***

***@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@***

***@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @***

***@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@***

***IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!***

***Someone could be eavesdropping on you right now (man-in-the-middle attack)!***

***It is also possible that the RSA host key has just been changed.***

***183***

***The fingerprint for the RSA key sent by the remote host is***

***41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.***

***Please contact your system administrator.***

***Add correct host key in /home/me/.ssh/known\_hosts to get rid of this message.***

***Offending key in /home/me/.ssh/known\_hosts:1***

***RSA host key for remote-sys has changed and you have requested strict***

***checking.***

***Host key verification failed.***

Це повідомлення з'являється в двох можливих ситуаціях. Перша: зловмисник міг спробувати провести атаку виду «зловмисник в середині». Це трапляється рідко, тому що всі знають, що ***ssh*** попереджає користувача про це. Більш ймовірна причина пов'язана з якимись змінами в віддаленій системі: наприклад, було виконане перевстановлення операційної системи або сервера **SSH**. Однак в інтересах безпеки не слід скидати з рахунків першу можливість. Завжди звертайтеся до системного адміністратора віддаленої системи, коли з'явиться це повідомлення.

Переконавшись в безневинність причин, що викликали це повідомлення, можна виправити проблему на стороні клієнта. Для цього за допомогою текстового редактора (наприклад, ***vim***) видаліть застарілий ключ з файлу ***~/.ssh/known\_hosts***. У прикладі повідомлення вище присутній рядок:

***Offending key in /home/me/.ssh/known\_hosts:1***

Це означає, що підозрюваний ключ зберігається в рядку 1, у файлі ***known\_hosts***. Видаліть цей рядок з файлу і дозвольте програмі ***ssh*** прийняти нові ідентифікаційні дані від віддаленої системи.

Крім відкриття сеансу командної оболонки в віддаленій системі ***ssh*** дозволяє також виконати єдину команду. Наприклад, у віддаленій системі **remote-sys** можна виконати команду ***free*** і отримати результати в локальній системі:

***$ ssh remote-sys free***

***me@twin4's password:***

***total used free shared buffers cached***

***Mem: 775536 507184 268352 0 110068 154596***

***-/+ buffers/cache: 242520 533016***

***Swap: 1572856 0 1572856***

Цей прийом відкриває можливість для досить цікавих варіантів використання, як в наступному прикладі, де висновок команди ls у віддаленій системі перенаправляється в локальний файл:

***$ ssh remote-sys 'ls \*' > dirlist.txt***

***me @ twin4's password:***

Зверніть увагу на одиничні лапки. Вони необхідні для запобігання підстановки шляху в локальній системі; нам потрібно, щоб підстановка була виконана в віддаленій системі. Аналогічно, якщо б нам було потрібно перенаправити висновок в файл у віддаленій системі, ми могли б помістити оператор перенаправлення і ім'я файлу всередину одиночних лапок:

***$ ssh remote-sys 'ls \* > dirlist.txt'***

***scp і sftp - безпечна передача файлів***

Пакет **OpenSSH** включає ще дві програми, здатні використовувати шифрований тунель **SSH** для копіювання файлів по мережі. Перша, ***scp*** (**secure copy** - безпечне копіювання) використовується для копіювання файлів, як вже знайома вам програма ***cp***. Основна відмінність полягає в необхідності випереджати шлях до вихідного або кінцевого файлу ім'ям віддаленого вузла і символом двокрапки за ним. Наприклад, скопіювати документ з ім'ям **document.txt** з домашнього каталогу в віддаленій системі **remote-sys** в поточний робочий каталог в локальній системі можна так:

***$ scp remote-sys:document.txt .***

***me@remote-sys's password:***

***document.txt 100% 5581 5.5KB/s 00:00***

За аналогією з командою ***ssh*** перед ім'ям віддаленого вузла можна вказати ім'я користувача, якщо ім'я облікового запису в віддаленій системі не збігається з ім'ям облікового запису в локальній системі:

***$ scp bob@remote-sys:document.txt .***

Друга програма копіювання файлів через SSH-з'єднання: ***sftp***. Як випливає з її імені - це безпечна заміна для програми ***ftp***. ***sftp*** діє практично так само, як оригінальна програма ***ftp***, яку ми використовували вище, тільки передає дані не у відкритому текстовому вигляді, а через шифрований тунель **SSH**. ***sftp*** має важливу перевагу перед звичайною програмою ***ftp*** - вона не вимагає, щоб на віддаленому вузлі працював сервер **FTP**. Їй необхідний тільки сервер **SSH**. Це означає, що будь-який комп'ютер, до якого можна підключитися за допомогою клієнта **SSH**, можна також використовувати в якості FTP-подібного сервера. Нижче наводиться приклад сеансу роботи з програмою ***sftp***:

***$ sftp remote-sys***

***Connecting to remote-sys...***

***me@remote-sys's password:***

***sftp> ls***

***ubuntu-8.04-desktop-i386.iso***

***sftp> lcd Desktop***

***sftp> get ubuntu-8.04-desktop-i386.iso***

***Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-desktop-i386.iso***

***/home/me/ubuntu-8.04-desktop-i386.iso 100% 699MB 7.4MB/s 01:35***

***sftp> bye***

**Лекція №10**

**«Регулярні вирази у Linux. Класи символів POSIX. Прості та розширені регулярні вирази POSIX»**

*Регулярні вирази* – це усього лиш символічна форма запису, що використовується для ідентифікації шаблонів у тексті. Вони, до певної міри, нагадують групові символи, використовувані командною оболонкою для вибору відповідних файлів та шляхів, але в ширшому масштабі. Регулярні вирази підтримуються багатьма інструментами командного рядка і більшістю мов програмування, щоб спростити вирішення задач, пов’язаних з обробкою тексту. Однак проблема в тому, що не всі регулярні вирази однакові; різні інструменти і мови програмування використовують власні «діалекти» регулярних виразів. Для цілей нашого обговорення ми обмежимося регулярними виразами, як вони визначені в стандарті POSIX (і підтримуються більшістю інструментів командного рядка) в противагу більшості мовам програмування (особливо це відноситься до Perl), де використовуються більш широкі та різноманітні форми запису.

***grep – пошук в тексті***

Назва ***grep***в дійсності пішло від фрази **«global regular expressions print»** (глобальний пошук за допомогою регулярного виразу і виведення), тобто, як бачите, ***grep*** має деяке відношення до регулярних виразів. В сутності, ***grep*** переглядає текстові файли в пошуку збігів з вказаним регулярним виразом і виводить в стандартний вивід усі рядки з такими збігами.

До сих пір ми передавали програмі ***grep*** фіксовані рядки, наприклад:

***$ ls /usr/bin | grep zip***

Ця команда виведе перелік усіх файлів із каталогу ***/usr/bin***, імена котрих містять підрядок ***zip***.

Програма ***grep*** має наступний синтаксис:

grep [*параметри*] *регулярний\_вираз* [*файл*...]

В табл. 10.1 перелічені найбільш використовувані параметри ***grep***.

Табл. 10.1.Параметри команди ***grep***

|  |  |
| --- | --- |
| **Параметр** | **Опис** |
| ***-i*** | Ігнорувати реєстр символів. Потребує не відрізняти символи верхнього і нижнього регістрів. Аналогічно працює параметр ***--ignore-case*** |
| ***-v*** | Інвертувати критерій. Зазвичай ***grep*** виводить усі рядки зі збігами. Цей параметр змушує ***grep*** виводити рядки, що не містять збігів. Аналогічно працює параметр ***--invert-match*** |
| ***-c*** | Вивести число збігів (або «незбігів») в присутності параметру ***–v*** замість самих текстових рядків. Аналогічно працює параметр ***--count*** |
| ***-l*** | Замість рядків зі збігами виводиться тільки імена файлів зі знайденими рядками. Аналогічно працює параметр ***--files-with-matches*** |
| ***-L*** | Діє подібно параметру ***–l***, але виводить тільки імена файлів, де не знайдено ні одного збігу. Аналогічно працює параметр ***--files-without-match*** |
| ***-n*** | На початку кожного рядка зі збігом виводить її номер в файлі. Аналогічно працює параметр ***--line-number*** |
| ***-h*** | Придушити вивід імені файлів при пошуку по декільком файлам. Аналогічно працює параметр ***--no-filename*** |

***Метасимволи і літерали***

Крім літералів регулярні вирази можуть містити **метасимволи***,* вони використовуються для визначення більш складних критеріїв зіставлення. До метасимволів регулярних виразів відносяться наступні символи:

***^$.[]{}-?\*+()|\***

Всі інші символи вважаються **літералами**. Втім, в деяких випадках символ зворотного слеша використовується для створювання **метапослідовностей***,* а також для екранування мета символів, щоб вони могли інтерпретуватися як літерали, а не як метасимволи.

***Будь-який символ***

Перший метасимвол, який ми розглянемо – це символ крапки (***.***), відповідний будь-якому символу в даній позиції. Наприклад:

***$ grep -h '.zip' dirlist\*.txt***

***bunzip2***

***bzip2***

***bzip2recover***

***gunzip***

***gzip***

***funzip***

***gpg-zip***

***preunzip***

***prezip***

***prezip-bin***

***unzip***

***unzipsfx***

Тут виконаний пошук в наших файлах збігів з регулярними виразами ***.zip***. В отриманих результатах мається пара важливих моментів, котрі необхідно відмітити. Зверніть увагу, що програма ***zip*** не була знайдена. Це пояснюється включенням в регулярний вираз метасимволу крапки, збільшивши довжину обов’язкового збігу до чотирьох символів; так як в імені програми ***zip*** всього три символи, вона не була знайдена. Крім того, якщо б в наших переліках мались імена файлів з розширенням ***.zip***, вони також не були б знайдені, бо символ крапки в розширенні файлу інтерпретувався б як «будь-який символ».

***Якоря***

Символ кришки (***^***) і знак долара (***$***) в регулярних виразах інтерпретуються як **якоря**. Це означає, що в їх присутності збіг із регулярним виразом можливий, тільки якщо збіг буде знайдено на початку рядка (***^***) або в його кінці(***$***).

***$ grep -h '^zip' dirlist\*.txt***

***zip***

***zipcloak***

***zipgrep***

***zipinfo***

***zipnote***

***zipsplit***

***$ grep -h 'zip$' dirlist\*.txt***

***gunzip***

***gzip***

***funzip***

***gpg-zip***

***preunzip***

***prezip***

***unzip***

***zip***

***$ grep -h '^zip$' dirlist\*.txt***

***Zip***

Тут виконується пошук в переліку файлів рядку zip, що знаходиться в початку рядка, в кінці рядка і займає весь рядок, від початку до кінця. Зверніть увагу, що регулярний вираз ***^$*** (початок і кінець без яких-небудь символів між ними) буде відповідати пустим рядкам.

***Вирази в квадратних дужках та класи символів***

В доповнення до можливостей описати в регулярному виразі збіги з будь-яким символом в заданій позиції за допомогою виразу в квадратних дужках можна також описати збіг з одним символом із певної множини. Вираз у квадратних дужках допомагає визначити безліч символів (включаючи символи, які інакше інтерпретувалися б як метасимволи), які знаходяться в даній позиції. У наступному прикладі використовується множина з двох символів, завдяки якому виявляються відповідності з послідовностями ***bzip*** і ***gzip***:

***$ grep -h '[bg]zip' dirlist\*.txt***

***bzip2***

***bzip2recover***

***gzip***

Множина може містити будь-яке число символів. Метасимволи, укладені в квадратні дужки, втрачають своє спеціальне значення. Лише два метасимвола інтерпретуються особливим чином, але при цьому вони мають інший зміст. Перший - символ кришки (^), який використовується для позначення заперечення; другий - дефіс (-), який використовується для позначення діапазону символів.

***Заперечення***

Якщо одразу після відкриваючої квадратної дужки стоїть символ кришки (^), інші символи множини інтерпретуються як неприпустимі в даній позиції. Перевіримо це, змінивши попередній приклад:

***$ grep -h '[^bg]zip' dirlist\*.txt***

***bunzip2***

***gunzip***

***funzip***

***gpg-zip***

***preunzip***

***prezip***

***prezip-bin***

***unzip***

***unzipsfx***

Включивши заперечення, ми отримали список файлів, імена яких містять послідовність ***zip***, якій передує будь-який символ, крім ***b*** або ***g***. Зверніть увагу, що файл ***zip*** не був знайдений. Символ заперечення не скасовує необхідність присутності символу в заданій позиції, він лише вимагає, щоб символ в цій позиції не належав зазначеній множині. Символ кришки позначає операцію заперечення, тільки якщо є першим символом в вираженні в квадратних дужках; в іншому випадку він втрачає своє спеціальне значення і перетворюється в звичайний символ.

***Традиційні діапазони символів***

Якщо необхідно сконструювати регулярний вираз, який знаходив би в наших списках всі файли з іменами, які починаються з великої літери, це можна виконати наступним чином:

***$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXZY]' dirlist\*.txt***

Досить просто помістити 26 букв у верхньому регістрі в вираз в квадратних дужках. Але необхідність введення всіх цих символів викликає деяке занепокоєння, тому передбачений інший спосіб:

***$ grep -h '^[A-Z]' dirlist\*.txt***

***MAKEDEV***

***ControlPanel***

***GET***

***HEAD***

***POST***

***X***

***X11***

***Xorg***

***MAKEFLOPPIES***

***NetworkManager***

***NetworkManagerDispatcher***

Ми скоротили множину з 26 буквами до 3-символьного діапазону. Так можна виразити будь-який діапазон символів і навіть кілька діапазонів, наприклад, для пошуку імен файлів, що починаються з букви або цифри:

***$ grep -h '^[A-Za-z0-9]' dirlist\*.txt***

Як випливає з прикладів, символ дефіса отримує в діапазонах спеціальне значення, тому виникає питання: як включити дефіс в вираз в квадратних дужках, щоб він інтерпретувався як звичайний символ? Для цього достатньо поставити його в початок виразу. наприклад:

***$ grep -h '[A-Z]' dirlist\*.txt***

Ця команда знайде всі імена файлів, що містять літери верхнього регістру. З іншого боку, наступний вираз:

***$ grep -h '[-AZ]' dirlist\*.txt***

Знайде усі імена файлів, що містять дефіс, букву А або букву Z.

***Класи символів POSIX***

Традиційні діапазони символів - простий і ефективний спосіб визначення наборів символів. На жаль, вони можуть використовуватися не з усіма програмами. Ми не мали жодних проблем з діапазонами, використовуючи програму ***grep***, але могли б зіткнутися з ними при використанні інших програм.

Можна використовувати діапазони символів майже так само, як вони використовуються в регулярних виразах, але є одна проблема:

***$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]\****

***/usr/sbin/MAKEFLOPPIES***

***/usr/sbin/NetworkManagerDispatcher***

***/usr/sbin/NetworkManager***

Ця команда повернула очікуваний результат - список імен файлів, що починаються з великої літери. Але наступна команда дасть зовсім інший результат (тут наведено лише частину результатів):

***$ ls /usr/sbin/[A-Z]\****

***/usr/sbin/biosdecode***

***/usr/sbin/chat***

***/usr/sbin/chgpasswd***

***/usr/sbin/chpasswd***

***/usr/sbin/chroot***

***/usr/sbin/cleanup-info***

***/usr/sbin/complain***

***/usr/sbin/console-kit-daemon***

За часів, коли операційна система UNIX тільки з'явилася на світ, був відомий тільки один набір символів - **ASCII**, і цей факт знайшов своє відображення в даній особливості. В **ASCII** перші 32 символи (з номерами 0-31) - це керуючі символи (такі, як табуляція, забій і повернення каретки). Наступні 32 (32-63) представляють друковані символи, включаючи більшість знаків пунктуації та цифри з нуля до дев'яти. Наступні 32 (з номерами 64-95) представляють літери верхнього регістру і кілька знаків пунктуації. Опираючись на цю класифікацію, системи, що використовують набір ASCII, притримувались наступної черги зіставлення:

***ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz***

Цей порядок відрізняється від лексикографічного, який виглядає ось так:

***aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ***

З ростом популярності UNIX за межами США виникла необхідність в підтримці символів, що не входять в алфавіт американської англійської мови. Таблиця **ASCII** була розширена до використання 8-бітних символів, і в неї додалися символи з номерами 128-255, використовувані в багатьох інших мовах. Для підтримки цієї можливості в стандарт POSIX було введено поняття **регіону (locale)**, що визначає вибір набору символів для конкретного географічного регіону. Дізнатися, яка мова налаштована у вашій системі, можна за допомогою команди:

***$ echo $LANG***

***en\_US.UTF-8***

При перевірці цього налаштування POSIX-сумісні програми використовують лексикографічний порядок, а не порядок проходження символів в наборі ASCII. Це пояснює поведінку команд, розглянуте вище. Коли діапазон символів ***[A-Z]*** інтерпретується в лексикографічному порядку, він включає всі алфавітні символи, крім символу ***a*** в нижньому регістрі, - саме це пояснює отриманий результат.

Для часткового вирішення цієї проблеми стандарт POSIX передбачає кілька класів символів, що описують діапазони символів. Вони перераховані в табл. 10.2.

Таблиця 10.2. Класи символів

|  |  |
| --- | --- |
| **Клас символів** | **Опис** |
| ***[:alnum:]*** | Алфавітно-цифрові символи, еквівалент діапазону ***[A-Za-z0-9]*** в **ASCII** |
| ***[:word:]*** | Те ж саме, що і ***[:alnum:]***, с додатковим символом підкреслення (***\_***) |
| ***[:alpha:]*** | Алфавітні символи; еквівалент діапазону ***[A-Za-z]*** в *ASCII* |
| ***[:blank:]*** | Включає символи проміжку і табуляції |
| ***[:cntrl:]*** | Керуючі символи **ASCII**: включає символи **ASCII** з кодами від 0 до 31 і 127 |
| ***[:digit:]*** | Цифри від 0 до 9 |
| ***[:graph:]*** | Символи що відображуються; включає символи **ASCII** з кодами від 33 до 126 |
| ***[:lower:]*** | Символи нижнього регістру |
| ***[:punct:]*** | Знаки пунктуації; еквівалент класу ***[-!"#$%&'()\*+,./:;<=>?@[\\\]\_`{|}~]*** в **ASCII** |
| ***[:print:]*** | Символи що печатаються; усі символи з класу ***[:graph:]*** і проміжок |
| ***[:space:]*** | Пробільні символи, включаючи пробіл, табуляцію, повернення каретки, перевід рядку, вертикальну табуляцію і переведення формату; еквівалент класу ***[\t\r\n\v\f]*** в **ASCII** |
| ***[:upper:]*** | Символи верхнього регістру |
| ***[:xdigit:]*** | Символи, що використовуються для представлення шістнадцятирічних цифр; еквівалент класу ***[0-9A-Fa-f]*** в **ASCII** |

***Прості і розширені регулярні вирази POSIX***

Якраз коли, здавалося б, проблема плутанини з діалектами регулярних виразів вирішена, виявляється, що стандарт POSIX також ділить реалізації регулярних виразів на два види: **прості регулярні вирази** (Basic Regular Expressions, BRE) і **розширені регулярні вирази** (Extended Regular Expressions, ERE) . Особливості, які розглядалися досі, підтримуються всіма POSIX-сумісними програмами і додатками, що реалізують BRE. Програма ***grep*** - одна з них.

Чим відрізняються **BRE** і **ERE**? Відмінності стосуються наборів метасимволів. У діалекті **BRE** підтримуються наступні мітки метасимволу: ***^$.[]\****. Всі інші вважаються літералами. У **ERE** в множину метасимволів (з відповідними їм функціями) додаються: ***(){}?+|***.

Однак (що найцікавіше) символи ***(){}*** інтерпретуються в **BRE** як метасимволи, якщо вони екрановані символом зворотного слеша, тоді як в **ERE** присутність зворотного слеша перед цими ж метасимволами перетворює їх в літерали.

Оскільки далі в цій главі ми розглянемо особливості, які є частиною **ERE**, необхідно використовувати іншу версію ***grep***. Традиційно діалект **ERE** підтримувався програмою ***egrep***, але GNU-версія ***grep*** також підтримує розширені регулярні вирази при виклику її з параметром ***-E***.

***Чергування***

Першою особливістю розширених регулярних виразів, яку ми обговоримо, буде чергування (alternation, або вираз вибору) - воно дозволяє вибирати збіг з одним з декількох виразів. Так само як вираження в квадратних дужках дозволяють одному символу відповідати безлічі вказаних символів, чергування дозволяє знаходити збіг з безліччю рядків або інших регулярних виразів.

Для демонстрації скористаємося комбінацією команд ***grep*** і ***echo***. Спочатку спробуємо виконати просте зіставлення рядків:

***$ echo "AAA" | grep AAA***

***AAA***

***$ echo "BBB" | grep AAA***

***$***

Досить простий приклад, в якому ми передаємо по конвеєру виведення команди ***echo*** на введення ***grep*** і бачимо результат. Якщо буде виявлено збіг, ми бачимо виведення; якщо збіг відсутній, нічого не виводиться.

Тепер додамо чергування, що позначається метасимволом вертикальної риски (***|***):

***$ echo "AAA" | grep -E 'AAA|BBB'***

***AAA***

***$ echo "BBB" | grep -E 'AAA|BBB'***

***BBB***

***$ echo "CCC" | grep -E 'AAA|BBB'***

***$***

Тут ми бачимо регулярний вираз '***AAA|BBB***', який означає «збіг з рядком ***AAA*** або з рядком ***BBB***». Так як це розширена особливість, ми додали в команду ***grep*** параметр ***-E*** (замість цього можна було б використовувати програму ***egrep***) і укласти регулярний вираз в лапки, щоб запобігти інтерпретацію командною оболонкою символу вертикальної риски як оператора конвеєра. У чергуванні може бути більше двох варіантів:

***$ echo "AAA" | grep -E 'AAA|BBB|CCC'***

***AAA***

Для об'єднання з іншими елементами регулярного виразу чергування можна укладати в круглі дужки ():

***$ grep -Eh '^(bz|gz|zip)' dirlist\*.txt***

Цьому висловом відповідатимуть імена файлів з наших списків, що починаються з ***bz***, ***gz*** або ***zip***. Якщо відкинути круглі дужки, сенс регулярного виразу зміниться, і йому будуть відповідати імена, що починаються з ***bz*** або містять ***gz*** або ***zip***:

***$ grep -Eh '^bz|gz|zip' dirlist\*.txt***

***Квантифікатори***

Розширені регулярні вирази підтримують кілька способів визначення числа збігів з елементом.

***?* - збіг з елементом нуль або один раз**

Цей квантіфікатор фактично означає: «збіг з попереднім елементом не обов'язковий». Уявіть, що потрібно перевірити допустимість номера телефону, і передбачається, що номер допустимий, якщо представлений в одній з двох форм: ***(nnn) nnn-nnnn*** або ***nnn nnn-nnnn***, де ***n*** - це цифра. Для перевірки можна було б використовувати наступний регулярний вираз:

***^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$***

У цьому виразі за круглими дужками слідують знаки питання, які вказують, що дужки можуть або бути відсутніми, або бути присутніми один раз. І знову, оскільки круглі дужки вважаються метасимволами (в **ERE**), ми екранували їх зворотними слешами, щоб вони інтерпретувалися як літерали.

Спробуємо застосувати цей вислів:

***$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]$'***

***(555) 123-4567***

***$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'***

***555 123-4567***

***$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'***

***$***

Тут регулярному виразу відповідають обидві форми запису номера телефону, але йому не відповідає номер, який містить нецифрові символи.

***\** - збіг з елементом нуль або більше разів**

Подібно метасимволу ***?***, зірочка (***\****) позначає необов'язковий елемент; проте, на відміну від знака питання (***?***), цей елемент може зустрічатися будь-яке число раз, а не тільки один раз. Уявіть, що нам потрібно перевірити, чи є рядок реченням. Щоб відповідати нашим вимогам рядок повинен починатися з великої літери, містити будь-яке число букв верхнього та нижнього регістру і пробілів і закінчуватися крапкою. Для пошуку збігів з цим (дуже приблизним) визначенням речення скористаємося наступним регулярним виразом:

***[[:upper:]][[:upper:][:lower:] ]\*\.***

Вираз складається з трьох елементів: вираз в квадратних дужках з класом символів ***[:upper:]***, вираз в квадратних дужках з двома класами символів, ***[:upper:]*** і ***[:lower:]***, і проміжком, і крапка, екранована зворотним слешем. Другий елемент супроводжується метасимволом ***\****, тому в нашому реченні йому може відповідати будь-яке число букв верхнього та нижнього регістру та пробілів, що слідують за першою літерою верхнього регістру:

***$ echo "This works." | grep -E '[[:upper:]][[:upper:][:lower:] ]\*\.'***

***This works.***

***$ echo "This Works." | grep -E '[[:upper:]][[:upper:][:lower:] ]\*\.'***

***This Works.***

***$ echo "this does not" | grep -E '[[:upper:]][[:upper:][:lower:] ]\*\.'***

***$***

Перші два приклади відповідають вислову, а третій - ні, тому що в ньому відсутній обов'язковий перший символ верхнього регістру і завершальна крапка.

***+* - збіг з елементом один або більше разів**

Метасимвол ***+*** діє майже так само, як ***\****, але вимагає збігу з попереднім елементом не менше одного разу. Наступному регулярному виразу відповідатимуть тільки рядки, що складаються з груп, які налічують один або кілька алфавітних символів і розділених одиночними проміжками:

***^([[:alpha:]]+ ?)+$***

Протестуємо його:

***$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'***

***This that***

***$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'***

***a b c***

***$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'***

***$ echo "abc d" | grep -E '^([[:alpha:]]+ ?)+$'***

***$***

Як бачите, цьому виразу не відповідає рядок "***a b 9***", тому що він містить неалфавітний символ; точно так же йому не відповідає рядок "***abc d***", тому що між символами ***c*** і ***d*** в ньому присутні більше одного пробілу.

***{}* - збіг з елементом певне число раз**

Метасимволи ***{*** і ***}*** використовуються, щоб висловити мінімальне і максимальне число обов'язкових збігів. Ці числа можна представити чотирма можливими способами, як показано в табл. 10.3.

Табл. 10.3. Визначення кількості збігів

|  |  |
| --- | --- |
| **Специфікатор** | **Значення** |
| ***{n}*** | Попередній елемент відповідає, якщо зустрічається ***n*** разів |
| ***{n,m}*** | Попередній елемент відповідає, якщо зустрічається не менш ніж ***n*** і не більш ніж ***m*** разів |
| ***{n,}*** | Попередній елемент відповідає, якщо зустрічається ***n*** або більше разів |
| ***{,m}*** | Попередній елемент відповідає, якщо зустрічається не більш ніж ***m*** разів |

Повертаючись до прикладу з телефонними номерами, ми скористаємося цим методом визначення повторень, щоб спростити вихідний регулярний вираз:

***^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$***

до

***^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$***

Протестуємо його:

***$ echo "(555) 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'***

***(555) 123-4567***

***$ echo "555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'***

***555 123-4567***

***$ echo "5555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$'***

***$***

Як бачите, змінена версія регулярного виразу успішно справляється з перевіркою номерів, з круглими дужками і без них, і відкидає неправильно оформлені номери.

***Практичні приклади застосування регулярних виразів***

Розглянемо кілька вже знайомих команд і подивимося, як вони можуть використовувати регулярні вирази.

***Перевірка списку телефонів за допомогою grep***

***$ cat phonelist.txt***

***(232) 298-2265***

***(624) 381-1078***

***(540) 126-1980***

***(874) 163-2885***

***(286) 254-2860***

***(292) 108-518***

***(129) 44-1379***

***(458) 273-1642***

***(686) 299-8268***

***(198) 307-2440***

Деякі номери оформлені неправильно, що дуже добре для цілей демонстрації їх перевірки за допомогою ***grep***.

Було б корисно просканувати файл в пошуку неприпустимих номерів і вивести їх.

***$ grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$' phonelist.txt***

***(292) 108-518***

***(129) 44-1379***

Тут ми використали параметр ***-v***, щоб обернути зіставлення і вивести тільки рядки, які не відповідають вказанному виразу.

Сам вираз містить якірні метасимволи на обох кінцях і тим самим гарантує відсутність додаткових символів зліва і праворуч від номера. Крім того, на відміну від прикладу, наведеного вище, цей вислів також вимагає обов'язкової наявності круглих дужок в номері.

***Пошук незвичайних імен файлів за допомогою find***

Команда ***find*** підтримує перевірку, засновану на регулярному виразі. Існує одна важлива обставина, яку слід пам'ятати, використовуючи регулярні вирази в командах ***find*** і ***grep***. Якщо ***grep*** виводить рядок, що містить збіг з регулярним виразом, то ***find*** вимагає точного збігу шляху з регулярним виразом. У наступному прикладі команда ***find*** використовує регулярний вираз для пошуку шляхів до файлів, що містять будь-які символи, що не входять в наступну множину:

***[-\_./0-9a-zA-Z]***

В результаті такого пошуку можна виявити імена файлів і каталогів, що містять проміжки та інші, потенційно шкідливі символи:

***$ find . -regex '.\*[^-\_./0-9a-zA-Z].\*'***

Через вимоги точного збігу всього шляху ми додали елемент. ***\**** з обох кінців виравиразу ження, що заміщає будь-яку кількість будь-яких символів (в тому числі і відсутність символів). В середині вираження знаходиться інвертований вираз в квадратних дужках, що містить множину символів, допустимих в іменах файлів і каталогів.

***Пошук файлів за допомогою locate***

Програма ***locate*** підтримує прості (параметр ***--regexp***) і розширені (параметр ***--regex***) регулярні вирази. Завдяки цьому можна виконувати ті ж операції, що проводилися вище з файлами ***dirlist***:

***$ locate --regex ‚bin/(bz|gz|zip)'***

***/bin/bzcat***

***/bin/bzcmp***

***/bin/bzdiff***

***/bin/bzegrep***

***/bin/bzexe***

***/bin/bzfgrep***

***/bin/bzgrep***

***/bin/bzip2***

***/bin/bzip2recover***

***/bin/bzless***

***/bin/bzmore***

***/bin/gzexe***

***/bin/gzip***

***/usr/bin/zip***

***/usr/bin/zipcloak***

***/usr/bin/zipgrep***

***/usr/bin/zipinfo***

***/usr/bin/zipnote***

***/usr/bin/zipsplit***

Використавши чергування, ми знайшли шляхи, що містять ***bin/bz****,* ***bin/gz***або***/bin/zip***.

**Лекція №11**

**«Робота з текстом у потоковому редакторі sed. Основні можливості редактору sed»**

Безумовно, однією з функцій, для виконання яких використовуються сценарії командного інтерпретатора, є робота з текстовими файлами. Сценарії командного інтерпретатора можуть допомогти не тільки при дослідженні файлів журналів, читанні файлів конфігурації і обробці елементів даних, а й при автоматизації трудомістких завдань маніпулювання даними будь-якого типу, що містяться в текстових файлах. Проте спроби маніпулювати вмістом текстових файлів з використанням тільки команд сценарію командного інтерпретатора можуть зажадати занадто багато трудовитрат. Тим користувачам, яким доводиться виконувати будь-які види маніпулювання даними в сценаріях командного інтерпретатора, необхідно ознайомитися з інструментами ***sed***і ***gawk***, передбаченими в Linux. Ці інструменти дозволяють істотно спростити будь-які завдання обробки даних, які тільки будуть потрібні.

**Редактор *sed***

Редактор ***sed*** відноситься до категорії потокових редакторів, на відміну від звичайного інтерактивного текстового редактора. В інтерактивному текстовому редакторі, такому як ***vim***, застосовуються клавіатурні команди, що дозволяють вставляти, видаляти або замінювати текст у даних в інтерактивному режимі. З іншого боку, потоковий редактор вносить зміни в потік даних на основі ряду правил, встановлених заздалегідь, до початку обробки даних редактором. Редактор ***sed*** дозволяє маніпулювати даними в потоці даних з урахуванням команд, заданих в командному рядку або збережені в текстовому файлі з командами. Цей редактор зчитує з пристрою введення одночасно по одній лінії даних, погоджує зміст рядка з даними, представленими в командах редактора, вносить дані зміни в режимі потокової обробки у відповідності з вказівками команд, а потім виводить знову сформовані дані в потік **STDOUT**. Після того як потоковий редактор зіставляє всі команди з рядком даних, відбувається перехід до наступного рядка даних і процес повторюється. По завершенні обробки потоковим редактором всіх рядків даних в потоці відбувається вихід з програми редактора. Оскільки команди застосовуються до даних послідовно, рядок за рядком, редактор ***sed*** повинен здійснювати тільки один прохід через потік даних для внесення всіх змін. Завдяки цьому редактор ***sed*** виконує свою роботу набагато швидше в порівнянні з інтерактивним редактором і дозволяє швидко, динамічно вносити зміни в дані у файлі даних. Команда sed має наступний формат:

***sed options script file***

Опції ***options***, для яких передбачені параметри дозволяють налаштувати поведінку команди ***sed***. До них відносяться опції, показані в табл. 11.1.

Табл. 11.1. Опції команди ***sed***

|  |  |
| --- | --- |
| **Параметр** | **Опис** |
| ***-e script*** | Додати команди, вказані в сценарії ***script***, до команд, що виконуються при обробці вхідних даних |
| ***-f file*** | Додати команди, наведені у файлі ***file***, до команд, що виконуються при обробці вхідних даних |
| ***-n*** | Не виробляти вихідні дані після виконання кожної команди, а чекати команду ***print*** |

Параметр ***script*** вказує окрему команду, яка застосовується до потокових даних. Якщо потрібно більше однієї команди, то слід скористатися або опцією ***-e*** для завдання команд у командному рядку, або опцією ***-f*** для визначення їх в окремому файлі. Для маніпулювання даними передбачені численні команди.

**Визначення однієї команди редактора в командному рядку**

За замовчуванням редактор ***sed***застосовує задану при його виклику команду до вхідного потоку **STDIN**. Це дозволяє передавати дані по каналу для обробки безпосередньо в редактор ***sed***. Нижче наведено короткий приклад, який показує, як виконується ця операція.

***$ echo "This is a test" | sed 's/test/big test/'***

***This is a big test***

У цьому прикладі використовується команда ***s*** редактора ***sed***. Команда ***s*** забезпечує підстановку другого рядка замість першого текстового рядка, заданого за допомогою шаблону між символами косою риси. У цьому прикладі відбувається підстановка слів ***big test*** замість слова***test***. Після запуску цього прикладу на виконання результати повинні відобразитися майже миттєво. У цьому швидкодії полягає одна з переваг редактора ***sed***. Він дозволяє зробити кілька операцій внесення змін у дані приблизно за такий же час, в яке деякі з інтерактивних редакторів можуть лише здійснити свій запуск. Безумовно, в цьому простому прикладі змінена тільки один рядок даних. Але настільки ж швидко результати формуються і при зміні великих файлів з даними:

***$ cat data1***

***The quick brown fox jumps over the lazy dog.***

***The quick brown fox jumps over the lazy dog.***

***The quick brown fox jumps over the lazy dog.***

***The quick brown fox jumps over the lazy dog.***

***$ sed 's/dog/cat/' data1***

***The quick brown fox jumps over the lazy cat.***

***The quick brown fox jumps over the lazy cat.***

***The quick brown fox jumps over the lazy cat.***

***The quick brown fox jumps over the lazy cat.***

Команда ***sed***виконує свою роботу і повертає дані майже миттєво. Результати відобразяться після обробки кожного рядка даних. Таким чином, перегляд отриманих даних може початися ще до того, як редактор ***sed*** завершить обробку всього файлу. Важливо враховувати, що редактор ***sed*** не змінює дані в самому текстовому файлі. Він лише передає змінений текст в потік **STDOUT**. Перегляд вихідного текстового файлу дозволяє переконатися в тому, що в ньому, як і раніше утримуються ті ж дані:

***$ cat data1***

***The quick brown fox jumps over the lazy dog.***

***The quick brown fox jumps over the lazy dog.***

***The quick brown fox jumps over the lazy dog.***

***The quick brown fox jumps over the lazy dog.***

**Використання декількох команд редактора в командному рядку**

Щоб викликати на виконання більше ніж одну команду з командного рядка ***sed***, досить скористатися опцією ***-e****:*

***$ sed -e 's/brown/green/; s/dog/cat/' data1***

***The quick green fox jumps over the lazy cat.***

***The quick green fox jumps over the lazy cat.***

***The quick green fox jumps over the lazy cat.***

***The quick green fox jumps over the lazy cat.***

Обидві команди застосовуються до кожного рядку даних у файлі. Команди повинні бути розділені крапкою з комою, а між кінцем команди і крапкою з комою не повинно бути ніяких прогалин. Замість крапок з комою для розділення команди можна викликати вторинне запрошення до вводу інформації в командному інтерпретаторі **bash**. Для цього достатньо ввести першу, відкриває одинарну лапку, щоб відкрити сценарій, після чого командний інтерпретатор bash продовжить запитувати введення таких команд, поки не буде введена закриваюча лапка:

***$ sed -e ' >***

***s/brown/green/ >***

***s/fox/elephant/ >***

***s/dog/cat/' data1***

***The quick green elephant jumps over the lazy cat.***

***The quick green elephant jumps over the lazy cat.***

***The quick green elephant jumps over the lazy cat.***

***The quick green elephant jumps over the lazy cat.***

Не можна забувати, що команда повинна бути завершена на тій же рядку, в якій є закриває одинарна лапка. Після того, як командний інтерпретатор **bash** виявляє закриває лапки, починається виконання самої команди. Після запуску команда ***sed***застосовує кожну задану команду до кожного рядка даних в текстовому файлі.

**Читання команд редактора файлу**

Нарешті, якщо потрібно забезпечити застосування до даних великої кількості команд ***sed***, простіше зберегти їх в окремий файл. Для вказівки файлу в команді ***sed*** застосовується опція ***-f****:*

***$ cat script1***

***s/brown/green/***

***s/fox/elephant/***

***s/dog/cat/***

***$ sed -f script1 data1***

***The quick green elephant jumps over the lazy cat.***

***The quick green elephant jumps over the lazy cat.***

***The quick green elephant jumps over the lazy cat.***

***The quick green elephant jumps over the lazy cat.***

В такому випадку не потрібно поміщати крапку з комою після кожної команди. Редактор ***sed*** виходить з того, що в кожному рядку файлу з командами міститься окрема команда. Як і при введенні команд у командному рядку, редактор ***sed*** зчитує команди з вказаного файлу і застосовує їх до кожного рядка у файлі даних.

**Основні відомості про редактор sed**

Ключем до успішного використання редактора ***sed*** є те, що при роботі з ним необхідно правильно задавати команди і формати, призначені для настройки процесу редагування тексту, кількість яких дуже велика. Далі наведено опис деяких з основних команд та засобів, які можна включити в сценарій для початку використання редактора ***sed***.

Вище вже було показано, як використовувати команду ***s*** (скорочення від ***substitute***) для вставки нового тексту замість існуючого в рядку. Але в команді ***substitute*** можна задавати кілька додаткових опцій, що дозволяють спростити обробку тексту.

**Прапори підстановки**

Застосовуючи команду ***substitute*** для вставки нового тексту замість існуючого тексту, виділені з шаблонами в текстовому рядку, необхідно дотримуватися обережності. Розглянемо, що відбувається в наступному прикладі:

***$ cat data2***

***This is a test of the test script.***

***This is the second test of the test script.***

***$ sed 's/test/trial/' data2***

***This is a trial of the test script.***

***This is the second trial of the test script.***

Команда ***substitute*** чудово справляється з заміною тексту в декількох рядках, але за замовчанням вона замінює тільки перше входження в кожному рядку. Щоб забезпечити обробку з допомогою команди ***substitute*** декількох входжень тексту, необхідно задати прапор підстановки. Прапор підстановки визначається після рядків команди підстановки:

***s/pattern/replacement/flags***

Передбачені прапори підстановки чотирьох наведених нижче типів:

* ***Число***. Вказує, до якого від початку рядка входження шаблону повинна бути застосована операція підстановки нового тексту замість старого.
* ***g***. Вказує, що новий текст необхідно підставити замість всіх входжень існуючого тексту.
* ***p***. Вказує, що необхідно вивести вміст вихідних рядків.
* ***w file***. Передбачає запис результатів підстановки в файл.

При підстановці першого типу можна вказати, замість якого входження шаблону зіставлення редактор ***sed*** повинен підставити новий текст:

***$ sed 's/test/trial/2' data2***

***This is a test of the trial script.***

***This is the second test of the trial script.***

Якщо в якості прапора підстановки буде задано число ***2***, редактор ***sed*** замінить текст, який визначається шаблоном, новим текстом, лише в другому входження шаблону в кожному рядку. Прапор підстановки g дозволяє виконати заміну для всіх входжень шаблону в тексті:

***$ sed 's/test/trial/g' data2***

***This is a trial of the trial script.***

***This is the second trial of the trial script.***

Прапор підстановки ***p*** виводить рядок, що містить шаблон зіставлення, який вказаний в команді ***substitute***. Цей прапор найчастіше застосовується в поєднанні з опцією ***-n*** редактора ***sed***:

***$ cat data3***

***This is a test line.***

***This is a different line.***

***$ sed -n 's/test/trial/p' data3***

***This is a trial line.***

Опція ***-n*** пригнічує виведення з редактора ***sed***. Тим не менш прапор підстановки ***p*** виводить всі рядки, які були змінені. Застосування поєднання цих двох прапорів призводить до формування таких результатів, в яких показано тільки рядки, змінені командою ***substitute***. Прапор підстановки ***w*** передбачає формування такого ж виведення, але отримані дані зберігаються у вказаному файлі

***$ sed 's/test/trial/w test' data3***

***This is a trial line.***

***This is a different line.***

***$ cat test***

***This is a trial line.***

Звичайні вихідні дані редактора ***sed*** надходять в потік **STDOUT**, але тільки рядки, які включають шаблон зіставлення, зберігаються у вказаному вихідному файлі.

**Заміна символів**

Іноді доводиться стикатися з тим, що в текстових рядках є символи, які неможливо задати в безпосередньому вигляді шаблону підстановки. У світі Linux одним з характерних прикладів подібних символів є коса риса. Кожен з таких символів необхідно супроводити екрануючим символом , в результаті, наприклад, для заміни імен шляхів у файлах доводиться застосовувати надмірно складні конструкції. Зокрема, припустимо, що у файлі ***/etc/passwd*** замість командного інтерпретатора **bash** необхідно вказати командний інтерпретатор **C**. Для цього можна виконати наступну команду:

***$ sed 's/\/bin\/bash/\/bin\/csh/' /etc/passwd***

Зазначимо, що коса риса використовується в якості розділювачів рядків, але цей символ можна встановити безпосередньо в тексті шаблону, тому кожному символу косою риси повинен передувати символ зворотної косої риски, який виконує роль екрануючого символу. Це часто призводить до плутанини і помилок. Щоб можна було простіше вирішити цю проблему, редактор ***sed*** передбачена можливість вибрати в команді ***substitute*** інший символ в якості визначає межі полів рядка:

***$ sed 's!/bin/bash!/bin/csh!' /etc/passwd***

В даному прикладі розділювачами полів рядка служить знак оклику, тому завдання читання та аналізу імен шляхів значно спрощується.

**Використання адрес**

По замовчуванню команди, установлені для виконання редактора ***sed***, застосовуються до всіх рядків текстових даних. Але іноді потрібно застосувати команду тільки до конкретного рядку або групі рядків, для чого призначена порядкова адресація. В редакторі ***sed*** передбачені дві форми порядкового адресації:

* завдання діапазону номерів рядків;
* застосування текстового шаблону, що дозволяє фільтрувати потрібні рядки.

В обох формах для вказівки адреси ***address*** використовується однаковий формат:

***[address]command***

Передбачена також можливість зібрати в групи кілька команд для застосування до рядків з конкретною адресою:

***address {***

***command1***

***command2***

***command3***

***}***

Редактор ***sed*** проводить обробку з допомогою заданих команд тільки тих рядків, які узгоджуються з вказаною адресою. В цьому розділі демонструється використання обох цих методів адресування в сценаріях редактора ***sed****.*

**Числова порядкова адресація**

Якщо використовується числова порядкова адресація, то для посилання на рядки служать дані про їх стан в текстовому потоці. Редактор ***sed*** присвоює першому рядку в текстовому потоці номер один, потім послідовно нарощує номери рядків після обробки кожного символу перекладу на новий рядок. Адресу, вказану в команді, може представляти собою номер єдиною рядка або діапазон номерів рядків, який заданий номер початкового рядка, кома і номер кінцевої рядка. Нижче наведено приклад визначення номера рядка, до якої повинна застосовуватися команда***sed***.

***$ sed '2s/dog/cat/' data1***

***The quick brown fox jumps over the lazy dog***

***The quick brown fox jumps over the lazy cat***

***The quick brown fox jumps over the lazy dog***

***The quick brown fox jumps over the lazy dog***

Зміни тексту внесені редактором ***sed*** тільки за вказаною адресою, у двох рядках, починаючи з рядка два. Нижче наведено ще один приклад, але на цей раз з використанням діапазону адрес рядків.

***$ sed '2,3s/dog/cat/' data1***

***The quick brown fox jumps over the lazy dog***

***The quick brown fox jumps over the lazy cat***

***The quick brown fox jumps over the lazy cat***

***The quick brown fox jumps over the lazy dog***

Якщо команду необхідно застосувати до групи рядків, що починається з деякої позиції в тексті й триває до кінця тексту, можна задати спеціальну адресу, знак долара:

***$ sed '2,$s/dog/cat/' data1***

***The quick brown fox jumps over the lazy dog***

***The quick brown fox jumps over the lazy cat***

***The quick brown fox jumps over the lazy cat***

***The quick brown fox jumps over the lazy cat***

Справа в тому, що не завжди відомо заздалегідь, яка кількість рядків даних в тексті, тому знак долара є дуже зручним.

**Використання фільтрів у вигляді текстових шаблонів**

Передбачений ще один метод скороченого позначення рядків, до яких застосовується команда, але він трохи складніше. Редактор ***sed*** дозволяє задавати текстовий шаблон , який використовується для фільтрації рядків, які призначені для обробки за допомогою команди. Для цього використовується наступний формат:

***/pattern/command***

Параметр ***pattern*** **(шаблон)** повинен бути заданий між символами косою риси. Редактор ***sed***буде застосовувати команду тільки рядків, у яких міститься вказаний текстовий шаблон. Наприклад, якщо потрібно змінити усталений командний інтерпретатор тільки для користувача ***Samantha***, можна використовувати команду ***sed***:

***$ grep Samantha /etc/passwd***

***Samantha:x:1001:1002:Samantha,4,,:/home/Samantha:/bin/bash***

***$ sed '/Samantha/s/bash/csh/' /etc/passwd***

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

***...***

***Samantha:x:1001:1002:Samantha,4,,:/home/Samantha:/bin/csh Timothy:x:1002:1005::/home/Timothy:/bin/sh   
Christine:x:1003:1006::/home/Christine:/bin/sh***

Очевидно, що команда була застосована лише до рядку, що містить зіставляється текстовий шаблон. Безумовно, застосування шаблону у вигляді рядка з незмінним значенням в деяких випадках є цілком задовільним, як у наведеному прикладі з ідентифікатором користувача, **userid**, але можливості такої конструкції дещо обмежені. Редактор ***sed*** дозволяє використовувати в текстових шаблонів засіб, зване регулярними виразами, завдяки чому з'являється можливість створювати досить складні шаблони. За допомогою регулярних виразів можна створювати розширені текстові шаблони, що представляють собою фактично формули зіставлення з даними будь-яких типів. У таких формулах застосовуються в поєднанні ряди символів знаків, спеціальних символів і фіксованих текстових символів для формування чіткого шаблону, який може бути правильно підібраний з урахуванням практично будь-якій ситуації обробки тексту. При роботі з регулярними виразами доводиться

**Групування команд**

Якщо потрібно виконати кілька команд, наведених в одному рядку, то слід згрупувати ці команди з використанням фігурних дужок. Редактор ***sed*** буде обробляти кожну команду, перераховану в рядках адреси:

***$ sed '2{***

***> s/fox/elephant/***

***> s/dog/cat/***

***> }' data1***

***The quick brown fox jumps over the lazy dog.***

***The quick brown elephant jumps over the lazy cat.***

***The quick brown fox jumps over the lazy dog.***

***The quick brown fox jumps over the lazy dog.***

Обидві команди виконуються стосовно до даної адреси. Крім того, безумовно, можна також вказувати діапазон адрес перед згрупованими командами

***$ sed '3,${***

***> s/brown/green/***

***> s/lazy/active/***

***> }' data1***

***The quick brown fox jumps over the lazy dog.***

***The quick brown fox jumps over the lazy dog.***

***The quick green fox jumps over the active dog.***

***The quick green fox jumps over the active dog.***

Редактор ***sed*** застосовує всі ці команди до всіх рядків в діапазоні адрес.

**Видалення рядків**

Команди, передбачені в редакторі ***sed***, не обмежуються лише командою підстановки одних текстових рядків замість інших. На той випадок, що доведеться видалити конкретні рядки тексту у потоці тексту, є команда ***delete***. Дія, що виконується командою ***delete*** (скорочено ***d***), в основному відповідає її назві. Ця команда видаляє всі текстові рядки, зіставляються з заданою схемою адресації. При використанні команди ***delete*** необхідно дотримуватися обережності, оскільки, якщо не буде задана схема адресації, відбудеться видалення всіх рядків з потоку:

***$ cat data1***

***The quick brown fox jumps over the lazy dog***

***The quick brown fox jumps over the lazy dog***

***The quick brown fox jumps over the lazy dog***

***The quick brown fox jumps over the lazy dog***

***sed 'd' data1***

Очевидно, що команда***delete*** стає найбільш зручною при використанні в поєднанні з вказаною адресою. З її допомогою можна видаляти конкретні рядки тексту з потоку даних або за номером рядка:

***$ sed '3d' data4***

***This is line number 1.***

***This is line number 2.***

***This is line number 4.***

або із зазначенням конкретного діапазону рядків:

***$ sed '2,3d' data4***

***This is line number 1.***

***This is line number 4.***

або з використанням спеціального символу кінця файлу:

***$ sed '3,$d' data7***

***This is line number 1.***

***This is line number 2.***

У команді ***delete*** може застосовуватися і засіб зіставлення з шаблонами редактора***sed***:

***$ sed '/number 1/d' data7***

***This is line number 2.***

***This is line number 3.***

***This is line number 4.***

Редактор ***sed*** видаляє рядок, що містить текст, який зіставляється з вказаним шаблоном.

**Увага!** Редактор ***sed*** не зачіпає вихідний файл. Всі видалені рядки зникають тільки у вихідних даних редактора ***sed***. Вихідний файл містить рядки, які користувач може вважати вже "видаленими".

Передбачена також можливість видалення діапазону рядків з використанням двох текстових шаблонів, але при цьому необхідно дотримуватися обережності. Перший заданий шаблон "включає" видалення рядків, а другий "вимикає" видалення рядків. Редактор ***sed*** видаляє всі рядки, що знаходяться між двома зазначеними рядками (в тому числі і самі зазначені рядки):

***$ sed '/1/,/3/d' data6***

***This is line number 4.***

Крім того, необхідно дотримуватися обережності і у зв'язку з тим, що це засіб видалення "включається" при кожному виявленні редактором ***sed*** початкового шаблону в потоці даних. Це може призвести до отримання непередбачених результатів:

***$ cat data5***

***This is line number 1.***

***This is line number 2.***

***This is line number 3.***

***This is line number 4.***

***This is line number 1 again.***

***This is text you want to keep.***

***This is the last line in the file.***

***$ sed '/1/,/3/d' data7***

***This is line number 4.***

Виявлення другого рядки, в якій міститься цифра 1, призвело до повторного виклику команди delete видалення решти рядків у потоці даних, оскільки кінцевий шаблон не був виявлений. Безумовно, ще одна очевидна проблема виникає, якщо задано такий кінцевий шаблон, який так і не виявляється в тексті:

***$ sed '/1/,/5/d' data8***

У даному випадку кошти видалення були "включені" при першому виявленні початкового зіставлення з шаблоном, а кінцеве зіставлення з шаблоном так і не було знайдено, тому відбулося видалення всього іншого потоку даних.

**Вставка і додавання тексту**

Як і слід було очікувати, редактор ***sed***, подібно будь-якому іншому редактору, дозволяє вставляти текстові рядки в потік даних і приєднувати до кінця потоку даних додаткові рядки. Вивчаючи відмінності між цими двома діями, необхідно враховувати наступне, щоб уникнути плутанини:

* команда ***insert*** (скорочено ***i***) додає символ переходу на новий рядок перед зазначеної рядком;
* команда ***append*** (скорочено ***a***) додає символ переходу на новий рядок після зазначеного рядка.

Причиною можливої плутанини при роботі з цими двома командами є їх формати. Задаючи команду вставки або додавання , не можна приводити саму команду і її дані в командному рядку. Рядок, що підлягає обміну або додавання, необхідно задавати окремо, на окремому рядку. Для цього використовується наступний формат:

***sed '[address]command\ new line'***

Текст, наведений у рядку ***new line***, з'являється у вихідних даних редактора ***sed*** в тому місці, яке зазначено в команді. Нагадаємо, що при використанні команди ***insert*** текст з'являється перед текстовим вмістом потоку даних:

***$ echo "Test Line 2" | sed 'i\Test Line 1'***

***Test Line 1***

***Test Line 2***

А при використанні команди ***append*** додається рядок з'являється після зазначеного тексту в потоці даних:

***$ echo "Test Line 2" | sed 'a\Test Line 1'***

***Test Line 2***

***Test Line 1***

Якщо робота з редактором ***sed*** виконується у запрошенні інтерфейсу командного рядка, то з'являється вторинне запрошення до вводу інформації, в якому повинна бути введена вставляється (або додається) рядок. Після введення нової рядки у цьому запрошенні необхідно також ввести останню частину команди редактора ***sed***. Слідом за тим, як вводиться закриває одинарна лапка, командний інтерпретатор bash приступає до обробки команди:

***$ echo "Test Line 2" | sed 'i\***

***> Test Line 1'***

***Test Line 1***

***Test Line 2***

Такий формат команди дозволяє легко додавати текст до або після всього заданого тексту в потоці даних, але іноді виникає необхідність додавання тексту в потоці даних. Щоб вставити або додати дані між рядками в потоці даних, необхідно використовувати адресування, щоб вказати редактора ***sed***, де повинні з'явитися дані. При використанні описаних вище команд кожен раз можна вказувати адресу тільки одного рядка. Для зіставлення з конкретної рядком можна застосовувати або числовий номер рядка, або текстовий шаблон, але можливість використання діапазону адрес не передбачена. І це обмеження цілком виправдано, оскільки не можна уявити собі, як можна зробити вставку або додавання, орієнтуючись на діапазон рядків, а не на окрему рядок. Нижче наведено приклад вставки нового рядка перед рядком 3 в потоці даних.

***$ sed '3i\***

***> This is an inserted line.' data6***

***This is line number 1.***

***This is line number 2.***

***This is an inserted line.***

***This is line number 3.***

***This is line number 4.***

А в наступному прикладі нова рядок додається після рядка 3 в потоці даних:

***$ sed '3a\***

***> This is an appended line.' data6***

***This is line number 1.***

***This is line number 2.***

***This is line number 3.***

***This is an appended line.***

***This is line number 4.***

При цьому здійснюється такий же процес, як і під час роботи команди ***insert***; відмінність полягає лише в тому, що вставлення нового рядка відбувається не до, а після рядка з зазначеним номером. Якщо потік даних включає багато рядків і потрібно приєднати новий рядок тексту до кінця потоку даних, то достатньо поставити знак долара ***$***, який представляє останній рядок даних:

***$ sed '$a\***

***> This is a new line of text.' data6***

***This is line number 1.***

***This is line number 2.***

***This is line number 3.***

***This is line number 4.***

***This is a new line of text.***

Той же принцип застосовується, якщо виникає необхідність додати новий рядок до початку потоку даних. Для цього достатньо передбачити вставку нового рядка перед рядком з номером один. Щоб вставити або додати кілька рядків тексту, необхідно вводити зворотну косу риску після кожного рядка знову заданого тексту, поки не буде досягнута остання текстовий рядок в тому місці, де потрібно вставити або додати текст:

***$ sed '1i\***

***> This is one line of new text.\***

***> This is another line of new text.' data6***

***This is one line of new text.***

***This is another line of new text.***

***This is line number 1.***

***This is line number 2.***

***This is line number 3.***

***This is line number 4.***

Обидві задані рядки додаються до потоку даних.

**Внесення змін у рядки**

Команда ***change*** дозволяє змінити вміст всієї рядка тексту в потоці даних. Ця команда має формат, аналогічний формату команд ***insert*** і ***append***, оскільки в ній знову вводиться рядок також повинна бути задана окремо від решти команди ***sed***:

***$ sed '3c\***

***> This is a changed line of text.' data6***

***This is line number 1.***

***This is line number 2.***

***This is a changed line of text.***

***This is line number 4.***

У цьому прикладі редактор ***sed*** змінює текст в рядку з номером ***3***. Для адресування рядка можна також використовувати текстовий шаблон:

***$ sed '/number 3/c\***

***> This is a changed line of text.' data6***

***This is line number 1.***

***This is line number 2.***

***This is a changed line of text.***

***This is line number 4.***

Команда ***change*** з цим текстовим шаблоном вносить зміни у всі рядки тексту в потоці даних, зіставлення з якими закінчується успішно.

***$ sed '/number 1/c\***

***> This is a changed line of text.' data7***

***This is a changed line of text.***

***This is line number 2.***

***This is line number 3.***

***This is line number 4.***

***This is a changed line of text.***

***This is yet another line.***

***This is the last line in the file.***

У команді ***change*** допускається застосування діапазону адрес, але отримані при цьому результати можуть виявитися не такими, як очікується:

***$ sed '2,3c\***

***> This is a new line of text.' data7***

***This is line number 1.***

***This is a new line of text.***

***This is line number 4.***

Замість зміни обох рядків з заданим текстом редактор *sed* застосував один і той самий заданий рядок тексту для заміни і того і іншого рядка.

**Лекція №12**

**«Додаткові можливості редактора sed. Програма рядкового розбору і обробки вхідного потоку gawk»**

***Додаткові можливості редактора sed***

***Команда transform***

Команда ***transform*** (скорочено ***y***) — це єдина команда редактора ***sed***, дія якої поширюється на окремі символи. У команді ***transform*** використовується наступний формат:

***[address]y/inchars/outchars/***

Команда ***transform*** виконує взаємно однозначне відображення значень символів ***inchars*** в значення символів ***outchars***. Перший символ з ***inchars*** перетворюється в перший символ з ***outchars***. Другий символ з ***inchars*** перетворюється в другий символ з ***outchars***. Таке відображення зазначених символів триває по всій довжині заданих рядків ***inchars*** і ***outchars***. Якщо рядки ***inchars*** і ***outchars*** не мають однакову довжину, то редактор ***sed*** формує повідомлення про помилку.

Нижче наведено простий приклад використання команди ***transform***.

***$ sed 'y/123/789/' data8***

***This is line number 7.***

***This is line number 8.***

***This is line number 9.***

***This is line number 4.***

***This is line number 7 again.***

***This is yet another line.***

***This is the last line in the file.***

Як показують результати виконання цієї команди, кожен примірник з числа символів, зазначених у шаблоні ***inchars***, був замінений символом з тієї ж позиції в шаблоні ***outchars***. Команда ***transform*** — це глобальна команда; іншими словами, з її допомогою автоматично виконується перетворення всіх символів, виявлених у текстовому рядку, не враховуючи того, в якій позиції знаходиться входження того чи іншого символу:

***$ echo "This 1 is a test of 1 try." | sed 'y/123/456/'***

***This 4 is a test of 4 try.***

Редактор ***sed*** перетворив у рядку обидва екземпляра символу ***1***, пов'язаного з шаблоном. Можливість обмежувати перетворення лише конкретними входженнями символів відсутня.

***Додаткові відомості про формування вихідних даних***

Для виведення інформації, що надходить з потоку даних, можуть також використовуватися наступні команди:

* команда ***p***, призначена для виведення текстових рядків;
* команда ***=***, з допомогою якої можна показати номери рядків;
* команда ***l***, призначена для виводу рядків з довільним вмістом.

***Виведення рядків***

Подібно до того, як діє прапор ***p*** в команді ***substitution***, команда ***p*** надсилає рядок у вихідний потік редактора ***sed***. Окремо узята, ця команда не виконує яких-небудь складних дій:

***$ echo "this is a test" | sed 'p'***

***this is a test***

***this is a test***

Всі результати її застосування зводяться до того, що у вихідний потік надходять текстові дані, відносно яких і без того відомо, що вони повинні бути присутніми у виведенні. Набагато частіше команда ***print*** (скорочено ***p***) застосовується для виведення рядків, які містять текст, пов'язаний з текстовим шаблоном:

***$ sed -n '/number 3/p' data7***

***This is line number 3.***

Використовуючи в командному рядку опцію ***-n***, можна придушити висновок всіх інших рядків і включити до складу результатів тільки такі рядки, які містять текст, пов'язаний з текстовим шаблоном. Команду в такому форматі можна також використовувати як зручний спосіб виведення в потік даних певної підмножини рядків:

***$ sed -n '2,3p' data7***

***This is line number 2.***

***This is line number 3.***

Команду ***print*** можна також використовувати, якщо необхідно бачити вміст рядка до внесення в неї змін, наприклад, за допомогою команди ***substitution*** або ***change***. ***Сценарій, в якому відображається рядок до її зміни, можна скласти наступним чином:***

***$ sed -n '/3/{***

***p***

***s/line/test/p***

***}' data7***

***This is line number 3.***

***This is test number 3.***

У цій команді редактора ***sed*** відбувається пошук рядка, що містить цифру ***3***, а потім виконуються дві команди. Перш за все, в сценарії використовується команда ***p*** для виведення вихідної версії рядка; потім у ньому для заміни тексту виконується команда s поряд з прапором p для висновку результуючого тексту. У виведенні команди відображаються і вихідний текст, і текст рядка.

***Виведення номерів рядків***

Команда ***=*** (знак рівності) виводить номер поточного рядка в потоці даних. Номери рядків збільшуються після виявлення кожного чергового символу позначення кінця рядка в потоці даних. Кожен раз, коли в потоці даних виявляється символ позначення кінця рядка, редактор ***sed*** обробляє цей символ як ознаку завершення рядка тексту:

***$ sed '=' data1***

***1***

***The quick brown fox jumps over the lazy dog.***

***2***

***The quick brown fox jumps over the lazy dog.***

***3***

***The quick brown fox jumps over the lazy dog.***

***4***

***The quick brown fox jumps over the lazy dog.***

Редактор ***sed*** виводить номер рядка перед самим текстовим вмістом рядка. Команда ***=*** може застосовуватися при пошуку конкретного текстового шаблону в потоці даних:

***$ sed -n '/number 4/{***

***=***

***p***

***}' data7***

***4***

***This is line number 4.***

Шляхом задання опції-n можна забезпечити, щоб редактор sed відображав тільки ті рядки, які містять текст, пов'язаний з шаблоном, а також вказував їх номери рядків.

**Виведення рядків з довільним вмістом**

Команда ***list*** (скорочено ***l***) дозволяє виводити в потік даних і текстові символи, і недруковані символи ASCII. Всі недруковані символи відображаються з використанням або їх вісімкових значень, яким передує зворотна коса риска, або стандартних позначень широко застосовуваних недрукованих символів у стилі мови ***C***, таких як позначення***\t***для знаків табуляції:

***$ cat data9***

***This line contains tabs.***

***$ sed -n 'l' data9***

***This\tline\tcontains\ttabs.$***

Якщо знаки табуляції служать для форматування тексту, то у відповідних позиціях також наводиться позначення ***\t***. Знак долара в кінці рядка вказує символ позначення кінця рядка . Якщо в потоці даних містяться екрануючі символи, то при використанні команди ***list*** відображення цих символів також відбувається із застосуванням восьмеричного коду:

***$ cat data10***

***This line contains an escape character***

***$ sed -n 'l' data10***

***This line contains an escape character \033[44m$***

У текстовому файлі data10 містяться екрануючі керуючі коди, призначені для зміни кольору дисплея. Якщо цей текстовий файл виводиться з використанням команди ***cat***, то такі екрануючі керуючі коди не відображаються, але відбувається їх обробка і, відповідно, змінюється колір на моніторі. А за допомогою команди ***list*** можна ознайомитися з тим, які екрануючі керуючі коди фактично задані у файлі. Наприклад, ***\033*** — це вісімкове значення коду **ASCII** для клавіші ***<Esc>***.

***Використання файлів при роботі з редактором sed***

Команда ***substitution*** підтримує прапори, що дозволяють працювати з файлами. У цих цілях можуть застосовуватися й інші команди редактора sed, які не пов'язані з необхідністю передбачати заміну тексту.

***Запис у файл***

Для запису рядків у файл призначена команда ***w***, яка має наступний формат:

***[address]w filename***

Значення імені файлу ***filename*** може бути задано з зазначенням відносного або повне ім'я шляху, але в будь-якому випадку користувач редактора ***sed*** повинен мати дозвіл на запис у файл. Для позначення адреси може застосовуватися метод адресування будь-якого типу, що використовується в редакторі ***sed***, будь то метод, заснований на вказівці єдиного номера рядка, текстового шаблону, діапазону номерів рядків або текстових шаблонів. Нижче наведено приклад, в якому відбувається виведення в текстовий файл тільки перших двох рядків з потоку даних.

***$ sed '1,2w test' data7***

***This is line number 1.***

***This is line number 2.***

***This is line number 3.***

***This is line number 4.***

***$ cat test This is line number 1.***

***This is line number 2.***

Зрозуміло, це не виключає можливість використовувати опцію ***-n*** команди ***sed***, щоб запобігти виведенню рядків в потік **STDOUT**.

Це - чудовий інструмент, який може застосовуватись, якщо необхідно створити файл даних з основного файлу на основі загальних текстових значень, на зразок тих, що містяться у списку розсилки:

***$ cat data11***

***Blum, Katie Chicago, IL***

***Mullen, Riley West Lafayette, IN***

***Snell, Haley Ft. Wayne, IN***

***Woenker, Matthew Springfield, IL***

***Wisecarver, Emma Grant Park, IL***

***$ sed -n '/IN/w INcustomers' data11***

***$ cat INcustomers Mullen, Riley***

***West Lafayette, IN Snell, Haley***

***Ft. Wayne, IN***

Редактор ***sed*** записує в кінцевий файл тільки рядки даних, які містять текстовий шаблон.

**Читання даних з файлу**

Вище вже було описано, як вставляти дані і додавати текст до потоку даних з командного рядка ***sed***. Команда ***read*** (скорочено ***r***) дозволяє вставляти дані, що містяться в окремому файлі. Команда ***read***має наступний формат:

***[address]r filename***

Параметр ***filename*** вказує абсолютне або відносне ім'я шляху для файлу, який містить дані. При використанні команди ***read***не передбачена можливість задавати діапазон адрес. Дозволяється задавати тільки єдиний номер рядка або адресу у вигляді текстового шаблону. Редактор ***sed*** вставляє текст з файлу після рядка, вказаною адресою.

***$ cat data12***

***This is an added line.***

***This is the second added line.***

***$ sed '3r data12' data7***

***This is line number 1.***

***This is line number 2.***

***This is line number 3.***

***This is an added line.***

***This is the second added line.***

***This is line number 4.***

При цьому редактор ***sed*** вставляє в потік даних текстові рядки з файлу даних. Така ж дія здійснюється при використанні адреси, заданого текстовим шаблоном:

***$ sed '/number 2/r data12' data7***

***This is line number 1.***

***This is line number 2.***

***This is an added line.***

***This is the second added line.***

***This is line number 3.***

***This is line number 4.***

Якщо текст повинен бути доданий до кінця потоку даних, достатньо лише скористатися в якості символічного позначення адреси знаком долара:

***$ sed '$r data12' data7***

***This is line number 1.***

***This is line number 2.***

***This is line number 3.***

***This is line number 4.***

***This is an added line.***

***This is the second added line.***

Чудовим додатком команди ***read*** є її використання в поєднанні з командою ***delete*** для заміни частини в файлі даних з іншого файлу. Наприклад, припустимо, що у текстовому файлі зберігається зразок листа, який виглядає приблизно таким чином:

***$ cat letter***

***Would the following people:***

***LIST***

***please report to the office.***

У цьому зразку списку використовується загальний заповнювач LIST замість списку адресатів. Щоб вставити дійсний список адресатів після заповнювача, залишається лише скористатися командою ***read***. Але при цьому у вихідних даних як і раніше залишається заповнює текст. Щоб видалити його, досить застосувати команду ***delete***. Результат виглядає наступним чином:

***$ sed '/LIST/{***

***> r data11***

***> d***

***>***

***}' letter***

***Would the following people:***

***Blum, Katie Chicago, IL***

***Mullen, Riley West Lafayette, IN***

***Snell, Haley Ft. Wayne, IN***

***Woenker, Matthew Springfield, IL***

***Wisecarver, Emma Grant Park, IL***

***please report to the office.***

Отже, заповнює текст замінений списком адресатів з файлу даних.

***Програма gawk***

Незважаючи на те, що редактор ***sed***являє собою досить зручний інструмент для оперативного внесення змін в текстові файли, він має свої обмеження. Часто для маніпулювання даними у файлі потрібно більш розвинутий інструмент, такий, який здатний надати середу, подібну середовищі звичайної мови програмування, що дозволяє модифікувати і реорганізовувати дані у файлі. Саме в таких обставинах може застосовуватися редактор ***gawk*** . Програма ***gawk*** являє собою версію GNU вихідної програми ***awk***, передбаченої в UNIX. Програма ***gawk*** дозволяє підняти на новий рівень складність виконуваних завдань порівняно з редактором ***sed***, оскільки в ній замість простих команд редактора застосовуються цілі сценарії на окремому мовою програмування. Мова програмування ***gawk*** дозволяє виконувати наступне:

* визначати змінні для зберігання даних;
* використовувати арифметичні та строкові операції для обробки даних;
* задавати структуровані програмні конструкції на зразок інструкцій if-then і циклів для забезпечення обробки даних за допомогою засобів перевірки умов;
* створювати відформатовані звіти, витягуючи елементи даних з файлу даних, а потім комбінуючи ці дані по-іншому, зі зміною послідовності розташування або формату.

Можливості формування звітів програми ***gawk*** часто використовуються для вилучення елементів даних з великих текстових файлів і подання даних у вигляді звіту, зручного для читання. Ідеальним прикладом вирішення цієї проблеми може слугувати форматування файлів журналів. Завдання пошуку причин порушення в роботі шляхом вивчення повідомлень про помилки у файлі журналу може виявитися складною. Програма gawk дозволяє вибрати з файлу журналу з допомогою фільтра тільки ті елементи даних, які становлять інтерес, а потім відформатувати їх таким чином, щоб читання важливих даних стало простіше.

***Формат команди gawk***

Програма gawk має наступний основний формат

***gawk options program file***

В табл. 12.1 наведено опції ***options***, підтримувані програмою ***gawk***. Опції командного рядка надають простий спосіб налаштування засобів, застосовуваних у програмі ***gawk***. Основна перевага програми ***gawk*** полягає в тому, що вона дозволяє використовувати програмні сценарії. Передбачена можливість написання сценаріїв для читання даних в текстовому рядку, а потім відображення даних і маніпулювання ними з метою створення вихідного звіту будь-якого типу.

Табл. 12.1. Опції***gawk***

|  |  |
| --- | --- |
| **Параметр** | **Опис** |
| ***-F fs*** | Надати файлу роздільник, який служить для розмежування полів даних в рядку |
| ***-f file*** | Вказати ім'я файлу, з якого повинна бути зчитана програма |
| ***-v var=value*** | Визначити змінну ***var*** і значення за замовчуванням ***value*** для використання в програмі ***gawk*** |
| ***-mf N*** | Задати максимальну кількість полів, що підлягають обробці у файлі даних |
| ***-mr N*** | Задати максимальний розмір запису у файлі даних |

**Читання програмного сценарію з командного рядка**

Сценарій програми ***gawk*** укладено в відкривають і закривають фігурні дужки. Між цими двома фігурними дужками повинні бути поміщені команди сценарію. Оскільки передбачається, що в командному рядку ***gawk*** весь сценарій представлений у вигляді одного текстового рядка, необхідно також укласти сценарій в одинарні лапки. Нижче наведено приклад простого сценарію програми gawk, що задається в командному рядку.

***$ gawk '{print "Hello John!"}'***

У цьому програмному сценарії визначена єдина команда – ***print***. Команда ***print*** виконує дію, що відповідає її назві: виводить текст в потік **STDOUT**. При спробі безпосередньо виконати цю команду можна випробувати певне розчарування, оскільки відразу ж після її введення нічого не відбувається. Справа в тому, що в цьому рядку не вказано ім'я файлу, тому програма ***gawk*** приступає до прийому даних з потоку **STDIN**. Після запуску програми на виконання просто відбувається перехід в режим очікування надходження тексту через **STDIN**. Після введення рядка і натискання клавіші **<ENTER>** програма ***gawk*** обробляє текст за допомогою програмного сценарію:

***$ gawk '{print "Hello John!"}'***

***This is a test***

***Hello John!***

***hello***

***Hello John!***

***This is another test***

***Hello John!***

Повністю аналогічно редактора ***sed***, програма ***gawk*** виконує програмний сценарій стосовно до кожної рядку тексту, наявної в потоці даних. Даний програмний сценарій налаштований на відображення постійно заданої текстового рядка, тому незалежно від тексту, введеного в потік даних, що відображаються одні і ті ж вихідні текстові дані. Для завершення роботи програми ***gawk*** необхідно подати сигнал про закінчення потоку даних. У командному інтерпретаторі **bash** передбачена комбінація клавіш для формування символу кінця файла **(End-of-File – EOF)** . Символ, який представляє собою ознака кінця файлу **EOF**, в командному інтерпретаторі **bash** формується з допомогою комбінації клавіш **<Ctrl>+<D>**. Після введення цієї комбінації клавіш робота програми ***gawk*** закінчується і відбувається повернення до запрошення інтерфейсу командного рядка.

***Використання змінних з позначенням полів даних***

Одна з найбільш важливих можливостей програми ***gawk*** полягає в тому, що вона дозволяє маніпулювати даними в текстовому файлі. Для цього в програмі передбачено автоматичне призначення змінної кожному елементу даних в рядку. За замовчуванням ***gawk*** призначає наступні змінні різних полів даних, виявлених у рядку тексту:

* ***$0*** представляє всю рядок тексту;
* ***$1*** являє перше поле даних у рядку тексту;
* ***$2*** являє друге поле даних у рядку тексту;
* ***$n*** є n-е поле даних в рядку тексту.

Для розмежування полів даних у текстовому рядку застосовується символ розділення полів. Після зчитування програмою ***gawk*** рядка тексту відбувається розподіл отриманого тексту на поля даних з використанням заданого символу розділення полів. За замовчуванням у програмі ***gawk*** в якості символів розділення полів розглядаються всі пробільні символи (такі як символи табуляції та пропуски). Нижче наведено приклад програми ***gawk***, яка зчитує текстовий файл і відображає тільки перше значення поля даних:

***$ cat data3***

***One line of test text.***

***Two lines of test text.***

***Three lines of test text.***

***$ gawk '{print $1}' data3***

***One***

***Two***

***Three***

У цій програмі використовується змінна поля ***$1*** для відображення першого поля з кожного рядка тексту. Якщо зчитується файл, в якому використовується інший символ розділення полів, такий можна вказати за допомогою опції ***-F***:

***$ gawk -F: '{print $1}' /etc/passwd***

***root***

***daemon***

***bin***

***sys***

***sync***

***games***

***man***

***lp***

***mail***

***...***

Ця коротка програма відображає перше поле даних у файлі паролів системи. У файлі ***/etc/passwd*** для розділення полів даних служить двокрапка, тому, щоб відокремити один від одного різні елементи даних, необхідно вказати двокрапка як символ розділення полів в опціях ***gawk***.

***Використання декількох команд в програмному сценарії***

Мова програмування була б занадто маловиразним, якщо з його допомогою можна було ставити в програмі тільки одну команду. Тому і в мові програмування ***gawk*** передбачена можливість комбінувати команди, створюючи повноцінну програму. Щоб скористатися декількома командами в програмному сценарії , заданому в командному рядку, досить розмежувати команди-крапкою з комою:

***$ echo "My name is Rich" | gawk '{$4="Christine"; print $0}'***

***My name is Christine***

Перша команда присвоює значення змінної поля ***$4***. Потім друга команда виводить вміст всього цього поля даних. Заслуговує уваги те, що у виведенні програми ***gawk*** четверте поле даних замінено у вихідному тексті новим значенням. Для введення команд програмного сценарію по одному рядку можна використовувати вторинне запрошення ***командного інтерпретатора:***

***$ gawk '{***

***> $4="testing"***

***> print $0 }'***

***This is not a good test.***

***This is not testing good test.***

Після завдання відкриває одинарних лапок командний інтерпретатор **bash** надає вторинне запрошення до вводу інформації, запитуючи продовження введення даних. Передбачена можливість додавати команди по одній в кожному рядку, поки не буде введена закриває одинарна лапка. Щоб вийти з програми, достатньо натиснути комбінацію клавіш ***<Ctrl>+<D>*** для вказівки на те, що на цьому закінчуються.

***Читання програми з файлу***

Як і у випадку редактора ***sed***, редактор ***gawk*** дозволяє зберігати програми у файлі і викликати їх з командного рядка:

***$ cat script2***

***{ print $1 "'s home directory is " $6 }***

***$ gawk -F: -f script2 /etc/passwd***

***root's home directory is /root***

***daemon's home directory is /usr/sbin ...***

***Samantha's home directory is /home/Samantha***

***Timothy's home directory is /home/Timothy***

***Christine's home directory is /home/Christine***

У програмному сценарії ***script2*** знову використовується команда ***print*** для виведення вмісту поля з даними про вихідному каталозі у файлі ***/etc/passwd*** (змінна поля ***$6***) і поле з даними про ідентифікатор користувача (змінна поля ***$1***). Передбачена можливість задавати в програмному файлі декілька команд. Для цього досить помістити кожну команду на окремому рядку. Крапки з комою використовувати не вимагається:

***$ cat script3***

***{***

***text = "'s home directory is "***

***print $1 text $6***

***}***

***$ gawk -F: -f script3 /etc/passwd***

***root's home directory is /root daemon's***

***home directory is /usr/sbin***

***...***

***Samantha's home directory is /home/Samantha***

***Timothy's home directory is /home/Timothy***

***Christine's home directory is /home/Christine***

У програмному сценарії ***script3*** визначена змінна, призначена для зберігання текстового рядка, використовуваної в команді ***print***. Заслуговує уваги те, що, на відміну від сценарію командного інтерпретатора, у програмі ***gawk*** для посилання на значення змінної не використовується знак долара.

***Виконання сценаріїв перед обробкою даних***

Програма***gawk*** дозволяє також вказувати, коли повинен бути виконаний програмний сценарій. За замовчуванням програма ***gawk*** зчитує рядок тексту з вхідного потоку, а потім виконує програмний сценарій стосовно до даних в рядку тексту. Іноді може знадобитися виконати сценарій перед обробкою даних, наприклад, щоб створити розділ з заголовком звіту. Для цієї мети використовується ключове слово ***BEGIN***. Команда ***BEGIN*** змушує програму ***gawk*** виконати фрагмент програмного сценарію, заданий після ключового слова ***BEGIN***, і лише потім наступає етап читання даних програмою ***gawk***:

***$ gawk 'BEGIN {print "Hello World!"}'***

***Hello World!***

На цей раз команда ***print*** відображає текст перед читанням будь-яких даних. Однак безпосередньо після відображення тексту відразу ж відбувається вихід з програми без очікування будь-яких даних. Причина цього полягає в тому, що ключове слово ***BEGIN*** вимагає лише виконання заданого фрагмента сценарію до початку обробки даних. Якщо потрібно провести обробку даних за допомогою звичайного програмного сценарію, то необхідно визначити для цього відповідну програму з допомогою ще одного розділу сценарію:

***$ cat data4***

***Line 1***

***Line 2***

***Line 3***

***$ gawk 'BEGIN { print "The data4 File Contents:" } { print $0 }' data4***

***The data4 File Contents:***

***Line 1***

***Line 2***

***Line 3***

Тепер після виконання сценарію ***BEGIN*** програмою ***gawk*** відбувається перехід до використання другого сценарію для обробки всіх необхідних даних з файлу. Застосовуючи подібні програми, необхідно дотримуватися обережності, оскільки всі розділи сценарію і раніше повинні бути задані у вигляді одного текстового рядка в командному рядку виклику програми ***gawk***. Відповідним чином повинні бути також розміщені одинарні лапки.

***Виконання сценаріїв після обробки даних***

Як і ключове слово ***BEGIN***, ключове слово ***END*** дозволяє задати фрагмент програмного сценарію, що виконується програмою ***gawk*** після читання даних:

***$ gawk 'BEGIN { print "The data4 File Contents:" } { print $0 } END { print "End of File" }' data4***

***The data4 File Contents:***

***Line 1***

***Line 2***

***Line 3***

***End of File***

Слідом за тим як програма ***gawk*** завершує висновок обробленого вмісту файлу, починається виконання команд у фрагменті сценарію ***END***. У цьому полягає чудова можливість додавання даних, нижнього колонтитула до звітів після завершення обробки всіх поточних даних. Зазначені елементи сценарію можуть бути зібрані воєдино як окремий, зручний у використанні файл програмного сценарію, призначеного для створення повного звіту з простого файлу даних:

***$ cat script4***

***BEGIN {***

***print "The latest list of users and shells"***

***print " Userid Shell"***

***print "-------- -------" FS=":"***

***}***

***{***

***print $1 " " $7"***

***}***

***END{***

***print "This concludes the listing"***

***}***

У цьому сценарії використовується розділ ***BEGIN*** для створення розділу заголовка звіту. У ньому також визначено спеціальна змінна ***FS***. У цьому полягає ще один спосіб визначення символу розділення полів. Завдяки цьому виключається залежність від того, чи буде користувач сценарію враховувати вимогу по визначенню символу розділення полів в опціях командного рядка. Нижче наведено трохи скорочений приклад виводу, отриманий при виконанні цього сценарію програми ***gawk***:

***$ gawk -f script4 /etc/passwd***

***The latest list of users and shells***

***Userid Shell***

***-------- ------***

***root /bin/bash***

***daemon /bin/sh***

***bin /bin/sh***

***...***

***Samantha /bin/bash***

***Timothy /bin/sh***

***Christine /bin/sh***

***This concludes the listing***

Як і слід було очікувати, розділ ***BEGIN*** створює текст заголовка, програмний сценарій обробляє інформацію із зазначеного файлу даних (***файл /etc/passwd***), а розділ ***END*** виводить текст нижнього колонтитула. Вивчаючи цей сценарій, можна хоч в якійсь мірі відчути, які можливості відкриваються при використанні навіть найпростіших сценаріїв ***gawk***.

**Лекція №13**

**«Введення до розробки програм на мовах C та С++ у ОС Linux. Компілятори gcc та g++. Використання утиліти make для компіляції програм»**

***Знайомство з компілятором GCC***

Засобами, традиційно використовуваними для створення програм для відкритих операційних систем, є інструменти розробника **GNU**. Зробимо маленьку історичну довідку. Проект **GNU** був заснований в 1984 році Річардом Столлманом. Його необхідність була викликана тим, що в той час співпраця між програмістами була утруднена, оскільки власники комерційного програмного забезпечення чинили численні перешкоди такій співпраці. Метою проекту **GNU** було створення комплекту програмного забезпечення під єдиною ліцензією, яка не допускала б можливості привласнення кимось ексклюзивних прав на це ПЗ. Частиною цього комплекту і є набір інструментів для розробника, яким ми будемо користуватися, і який повинен входити в усі дистрибутиви Linux.

Одним з цих інструментів є компілятор **GCC**. Спочатку ця абревіатура розшифровувалась, як **GNU C Compiler**. Зараз вона означає - GNU Compiler Collection.

Створимо першу програму за допомогою **GCC**. За традицією, що склалася, перша програма буде просто виводити в консолі вітання ***«Hello world!»***.

Файли з вихідними кодами програм, які ми будемо створювати, це звичайні текстові файли, і створювати їх можна за допомогою будь-якого текстового редактора (наприклад **GEdit** **KWrite**, Kate, а також більш традиційні для користувачів Linux - ***vi*** і ***emacs***). Крім текстових редакторів, існують спеціалізовані середовища розробки зі своїми вбудованими редакторами. Одним з таких засобів є **KDevelop**. Цікаво, що в ньому є вбудований редактор і вбудована консоль, розташована прямо під редактором. Так що можна прямо в одній програмі, не перемикаючись між вікнами, і редагувати код і давати консольні команди.

Створимо окремий каталог ***hello***. Це буде каталог нашого першого проекту. У ньому створимо текстовий файл ***hello.c*** з наступним текстом:

***#include <stdio.h>***

***int main(void)***

***{***

***printf("Hello world!\n");***

***return(0);***

***}***

Потім в консолі зайдемо в каталог проекту і наберемо команду

***$ gcc hello.c***

Тепер подивимось уважно, що сталося. В каталозі з'явився новий файл ***a.out***. Це і є виконуваний файл. Запустимо його. Наберемо в консолі:

***$ ./a.out***

Програма повинна запуститися, тобто повинен з'явитися текст:

***Hello world!***

Компілятор ***gcc*** за замовчуванням присвоює всім створеним виконуваних файлів ім'я ***a.out***. Якщо хочете назвати його по-іншому, потрібно до команди на компіляцію додати прапор ***-o*** і ім'я, яким ви хочете його назвати.

Ви, звичайно, звернули увагу, що, коли ми запускаємо програму з нашого каталогу розробки, ми перед назвою файлу набираємо точку і слеш. Навіщо ж ми це робимо?

Справа в тому, що, якщо ми наберемо тільки назва виконуваного файлу, операційна система буде шукати його в каталогах ***/usr/bin*** і /***usr/local/bin***, і, природно, не знайде. Каталоги ***/usr/bin*** і ***/usr/local/bin*** - системні каталоги розміщення виконуваних програм. Перший з них призначений для розміщення стабільних версій програм, як правило, входять в дистрибутив Linux. Другий - для програм, що встановлюються самим користувачем (за стабільність яких ніхто не ручається). Така система потрібна, щоб відокремити їх один від одного. За замовчуванням при складанні програми встановлюються в каталог ***/usr/local/bin***. Вкрай небажано поміщати щось зайве в ***/usr/bin*** або видаляти щось звідти вручну, тому що це може привести до краху системи. Там повинні розміщуватися програми, за стабільність яких відповідають розробники дистрибутива.

Щоб запустити програму, що знаходиться в іншому місці, треба прописати повний шлях до неї, наприклад так:

***/home/dima/projects/hello/hello***

Або інший варіант: прописати шлях щодо поточного каталогу, в якому ви в даній момент перебуваєте в консолі. При цьому одна точка означає поточний каталог, дві точки - батьківський. Наприклад, команда ***./hello*** запускає програму **hello**, що знаходиться в поточному каталозі, команда ***../hello*** - програму **hello**, що знаходиться в батьківському каталозі, команда ***./projects/hello/hello*** - програму у вкладених каталогах, які перебувають у поточному.

***$ gcc hello.c -o hello***

Ми бачимо, що в каталозі з'явився виконуваний файл з назвою hello. Запустимо його.

***./hello***

Як бачите, вийшов такий самий виконуваний файл, тільки зі зручною для нас назвою.

Прапор ***-o*** є лише одним з численних прапорів компілятора ***gcc***. Щоб переглянути всі можливі прапори, можна скористатися довідковою системою ***man***.

Тепер розглянемо, що ж робить програма ***gcc***. Її робота включає три етапи: обробка препроцесором, компіляція і компонування (або лінковка).

Препроцесор включає в основний файл вміст всіх заголовних файлів, зазначених в директивах ***#include***. У заголовних файлах зазвичай знаходяться оголошення функцій, використовуваних в програмі, але не визначених у тексті програми. Їх визначення знаходяться десь в іншому місці: або в інших файлах з вихідним кодом або в бінарних бібліотеках.

Друга стадія - компіляція. Вона полягає в перетворенні тексту програми на мові **C/C++** в набір машинних команд. Результат зберігається в об'єктному файлі. Зрозуміло, на машинах з різною архітектурою процесора виконавчі файли виходять в різних форматах, і на одній машині неможливо запустити бінарний файл, зібраний на іншій машині (хіба тільки, якщо у них однакова архітектура процесора і однакові операційні системи). Ось чому програми для UNIX-подібних систем поширюються в вигляді вихідних кодів: вони повинні бути доступні всім користувачам, незалежно від того, у кого який процесор і яка операційна система.

Остання стадія - компоновка. Вона полягає в зв'язуванні всіх об'єктних файлів проекту в один, зв'язуванні викликів функцій з їх визначеннями, і приєднанням бібліотечних файлів, що містять функції, які викликаються, але не визначені в проекті. В результаті формується файл, що запускається - наша кінцева мета. Якщо якась функція в програмі використовується, але компонувальник не знайде місце, де ця функція визначена, він видасть повідомлення про помилку, і відмовиться створювати виконуваний файл.

Тепер подивимося на практиці, як все це виглядає. Напишемо іншу програму. Це буде примітивний калькулятор, здатний додавати, віднімати, множити і ділити. При запуску він буде запитувати по черзі два числа, над якими слід провести дію, а потім зажадає ввести знак арифметичної дії. Це можуть бути чотири знаки: «***+***», «***-***», «***\****», «***/***». Після цього програма виводить результат і зупиняється (повертає нас в операційну систему, а точніше - в командний інтерпретатор, з якого ми програму і викликали).

Створимо для проекту нову папку ***kalkul***, в ній створимо файл ***kalkul.c***.

***#include <stdio.h>***

***int main(void)***

***{***

***float num1;***

***float num2;***

***char op;***

***printf("Перше число: ");***

***scanf("%f",&num1);***

***printf("Друге число: ");***

***scanf("%f",&num2);***

***printf("Оператор ( + - \* / ): ");***

***while ((op = getchar()) != EOF)***

***{***

***if (op == '+')***

***{***

***printf("%6.2f\n",num1 + num2);***

***break;***

***}***

***else if (op == '-')***

***{***

***printf("%6.2f\n",num1 - num2);***

***break;***

***}***

***else if (op == '\*')***

***{***

***printf("%6.2f\n",num1 \* num2);***

***break;***

***}***

***else if (op == '/')***

***{***

***if (num2 == 0)***

***{***

***printf("Помилка: ділення на нуль!\n");***

***break;***

***}***

***else***

***{***

***printf("%6.2f\n",num1 / num2);***

***break;***

***}***

***}***

***}***

***return 0;***

***}***

Отже, перш за все, як було сказано, виконується препроцессінг. Для того, щоб подивитися, що на цьому етапі робиться, скористаємося опцією ***-E***. Ця опція зупиняє виконання програми на етапі обробки препроцесором. В результаті виходить файл вихідного коду з включеним в нього вмістом заголовків файлів.

У нашому випадку ми включали один заголовки - ***stdio.h*** - колекцію стандартних функцій вводу-виводу. Ці функції і виводили на консоль потрібний текст, а також зчитували з консолі вводяться нами слова.

Введемо наступну команду:

***$ gcc -E kalkul.c -o kalkul.cpp***

Отриманому файлу ми дали ім'я ***kalkul.cpp***. Відкрийте його. Зверніть увагу на те, що він досить довгий. Це тому що в нього увійшов весь код заголовки ***stdio.h***. Крім того, препроцесор сюди додав деякі теги, що вказують компілятору спосіб зв'язку з оголошеними функціями. Основний текст нашої програми видно тільки в самому низу.

Можна заодно подивитися, які ще функції оголошені в заголовному файлі stdio.h. Якщо вам захочеться отримати інформацію про який-небудь функції, можна поцікавитися про неї у вбудованому керівництві ***man***. Наприклад, якщо вам раптом захочеться дізнатися, що ж робить функція ***fopen***, можна набрати:

***$ man fopen***

Багато інформації також є в довідковій системі info.

***$ info fopen***

Можна поцікавитися і всім заголовним файлом відразу.

***$ man stdio.h***

***$ info stdio.h***

Подивимося тепер наступний етап. Створимо об'єктний файл. Об'єктний файл є «дослівний» переклад нашого програмного коду на машинну мову, поки без зв'язку викликаються функцій з їх визначеннями. Для формування об'єктного файлу служить опція ***-c***.

***$ gcc -c kalkul.c***

Назва одержуваного файлу можна не вказувати, так як компілятор просто бере назву вихідного і змінює розширення ***.c*** на ***.o*** (вказати можна, якщо нам захочеться назвати його по-іншому).

Якщо ми створюємо об'єктний файл з файлу, вже обробленого препроцесором (наприклад, такого, який ми отримали вище), то ми повинні обов'язково вказати явно, що компільований файл є файлом вихідного коду, оброблений препроцесором, і має теги препроцесора. В іншому випадку він буде оброблятися, як звичайний файл **C++**, без урахування тегів препроцесора, а значить зв'язок з оголошеними функціями не буде встановлено. Для явної вказівки на мову і формат оброблюваного файлу служить опція ***-x***. Файл **C++**, оброблений препроцесором позначається ***cpp-output***.

***$ gcc -x cpp-output -c kalkul.cpp***

Нарешті, останній етап - компоновка. Отримуємо з виконуваного об'єктного файлу.

***$ gcc kalkul.o -o kalkul***

Можна його запускати.

***./kalkul***

Cправжні програми дуже рідко складаються з одного файлу. Як правило вихідних файлів кілька, і вони об'єднані в проект. І в деяких виняткових випадках програму доводиться компонувати з декількох частин, написаних на різних мовах. У цьому випадку доводиться запускати компілятори різних мов, щоб кожен отримав об'єктний файл зі свого вихідного файлу, а потім вже ці отримані об'єктні файли компонувати в виконувану програму.

***Приклад проекту з декількох файлів***

Напишемо тепер програму, що складається з двох вихідних файлів і одного заголовку. Для цього візьмемо наш калькулятор і переробимо його. Тепер після введення першого числа треба відразу вводити дію. Якщо дію оперує тільки з одним числом (як у випадку синуса, косинуса, тангенса, квадратного кореня), результат відразу буде виведений. Якщо знадобиться друге число, воно буде спеціально запитуватися.

Створимо каталог проекту ***kalkul2***. У ньому створимо три файли: ***calculate.h***, ***calculate.c***, ***main.c***.

Файл ***calculate.h***:

***#ifndef CALCULATE\_H\_***

***#define CALCULATE\_H\_***

***float Calculate(float Numeral, char Operation[4]);***

***#endif /\*CALCULATE\_H\_\*/***

Файл ***calculate.c***:

***#include <stdio.h>***

***#include <math.h>***

***#include <string.h>***

***#include "calculate.h"***

***float Calculate(float Numeral, char Operation[4])***

***{***

***float SecondNumeral;***

***if(strncmp(Operation, "+", 1) == 0)***

***{***

***printf("Другий доданок: ");***

***scanf("%f",&SecondNumeral);***

***return(Numeral + SecondNumeral);***

***}***

***else if(strncmp(Operation, "-", 1) == 0)***

***{***

***printf("Від'ємник: ");***

***scanf("%f",&SecondNumeral);***

***return(Numeral - SecondNumeral);***

***}***

***else if(strncmp(Operation, "\*", 1) == 0)***

***{***

***printf("Добуток: ");***

***scanf("%f",&SecondNumeral);***

***return(Numeral \* SecondNumeral);***

***}***

***else if(strncmp(Operation, "/", 1) == 0)***

***{***

***printf("Дільник: ");***

***scanf("%f",&SecondNumeral);***

***if(SecondNumeral == 0)***

***{***

***printf("Помилка: Ділення на нуль! ");***

***return(HUGE\_VAL);***

***}***

***else***

***return(Numeral / SecondNumeral);***

***}***

***else if(strncmp(Operation, "pow", 3) == 0)***

***{***

***printf("Ступінь: ");***

***scanf("%f",&SecondNumeral);***

***return(pow(Numeral, SecondNumeral));***

***}***

***else if(strncmp(Operation, "sqrt", 4) == 0)***

***return(sqrt(Numeral));***

***else if(strncmp(Operation, "sin", 3) == 0)***

***return(sin(Numeral));***

***else if(strncmp(Operation, "cos", 3) == 0)***

***return(cos(Numeral));***

***else if(strncmp(Operation, "tan", 3) == 0)***

***return(tan(Numeral));***

***else***

***{***

***printf("Неправильно введена дія ");***

***return(HUGE\_VAL);***

***}***

***}***

Файл ***main.c***:

***#include <stdio.h>***

***#include "calculate.h"***

***int main(void)***

***{***

***float Numeral;***

***char Operation[4];***

***float Result;***

***printf("Число: ");***

***scanf("%f",&Numeral);***

***printf("Арифметична дія (+,-,\*,/,pow,sqrt,sin,cos,tan): ");***

***scanf("%s",&Operation);***

***Result = Calculate(Numeral, Operation);***

***printf("%6.2f\n",Result);***

***return 0;***

***}***

У нас є два файли вихідного коду (c-файли) і один заголовний (h-файл). Заголовний включається в обидва c-файли.

Скомпілюємо ***calculate.c***:

***$ gcc -c calculate.c***

Отримали ***calculate.o***. Тепер те саме зробимо з файлом ***main.c***:

***$ gcc -c main.c***

Тепер треба з цих двох об'єктних файлів зробити файл, що запускається:

***$ gcc calculate.o main.o -o kalkul***

Однак при цьому у консолі буде виведено повідомлення про помилку:

***calculate.o(.text+0x1b5): In function `Calculate':***

***calculate.c: undefined reference to `pow'***

***calculate.o(.text+0x21e):calculate.c: undefined reference to `sqrt'***

***calculate.o(.text+0x274):calculate.c: undefined reference to `sin'***

***calculate.o(.text+0x2c4):calculate.c: undefined reference to `cos'***

***calculate.o(.text+0x311):calculate.c: undefined reference to `tan'***

***collect2: ld returned 1 exit status***

***Undefined reference*** означає посилання на функцію, яка не визначена. В даному випадку ***gcc*** не знайшов визначення функцій ***pow***, ***sqrt***, ***sin***, ***cos***, ***tan***. Де ж їх знайти?

Як вже говорилося раніше, визначення функцій можуть перебувати в бібліотеках. Це скомпільовані виконавчі файли, що містять колекції однотипних операцій, які часто викликаються з багатьох програм, а тому немає сенсу багаторазово писати їх код в програмах. Стандартне розташування файлів бібліотек - каталоги ***/usr/lib*** і ***/usr/local/lib*** (при бажанні можна додати шлях). Якщо бібліотечний файл має розширення ***.a***, то це статична бібліотека, тобто при компонуванні весь її двійковий код включається в виконуваний файл. Якщо розширення ***.so***, то це динамічна бібліотека. Це означає в виконуваний файл програми поміщається тільки посилання на бібліотечний файл, а вже з нього і запускається функція.

Коли ми писали програму ***hello***, ми використовували функцію ***printf*** для виведення текстового рядка. Однак, як ви пам'ятаєте, ми ніде не писали визначення цієї функції. Звідки ж вона тоді викликається?

Просто при компонуванні будь-якої програми компілятор ***gcc*** за замовчуванням включає в файл, що запускається бібліотеку ***libc***. Це стандартна бібліотека мови C. Вона містить рутинні функції, необхідні абсолютно у всіх програмах, написаних на C, в тому числі і функцію ***printf***. Оскільки бібліотека ***libc*** потрібна у всіх програмах, вона включається за замовчуванням, без необхідності давати окреме зазначення на її включення.

Решту бібліотек треба вимагати включати явно. Адже не можна ж в усі програми поміщати абсолютно все бібліотеки. Тоді виконуваний файл роздується до немислимо великих розмірів. Одним програмами потрібні одні функції, іншим - інші.

Нам в даному випадку потрібна бібліотека ***libm***. Саме вона містить всі основні математичні функції. Вона вимагає включення в текст програми заголовки ***<math.h>***.

Крім цього дистрибутиви Linux містять і інші бібліотеки, наприклад:

* ***libGL*** – виведення тривимірної графіки в стандарті **OpenGL**. Потрібно заголовки ***<GL/gl.h>***.
* ***libcrypt*** – криптографічні функції. Потрібно заголовки ***<crypt.h>***.
* ***libcurses*** – псевдографіка в символьному режимі. Потрібно заголовки ***<curses.h>***.
* ***libform*** – створення екранних форм в текстовому режимі. Потрібно заголовки ***<form.h>***.
* ***libgthread*** – підтримка багатопотокового режиму. Потрібно заголовки ***<glib.h>***.
* ***libgtk*** – графічна бібліотека в режимі X Window. Потрібно заголовки ***<gtk/gtk.h>***.
* ***libhistory*** – робота з журналами. Потрібно заголовки ***<readline/readline.h>***.
* ***libjpeg*** – робота з зображенням у форматі JPEG. Потрібно заголовки ***<jpeglib.h>***.
* ***libncurses*** – робота з псевдографікою в символьному режимі. Потрібно заголовки ***<ncurses.h>***.
* ***libpng*** – робота з графікою в форматі PNG. Потрібно заголовки ***<png.h>***.
* ***libpthread*** – багатопотокова бібліотека POSIX. Стандартна багатопотокова бібліотека для Linux. Потрібно заголовки ***<pthread.h>***.
* ***libreadline*** – робота з командним рядком. Потрібно заголовки ***<readline/readline.h>***.
* ***libtiff*** – робота з графікою в форматі TIFF. Потрібно заголовки ***<tiffio.h>***.
* ***libvga*** – низькорівнева робота з VGA і SVGA. Потрібно заголовки ***<vga.h>***.

А також багато-багато інших.

Зверніть увагу, що назви всіх цих бібліотек починаються з буквосполучення ***lib-***. Для їх явного включення в виконуваний файл, потрібно додати до команди ***gcc*** опцію ***-l***, до якої разом додати назву бібліотеки без ***lib-***. Наприклад, щоб включити бібліотеку ***libvga*** треба вказати опцію ***-lvga***.

Нам потрібні математичні функції ***pow***, ***sqrt***, ***sin***, ***cos***, ***tan***. Вони, як вже було сказано, знаходяться в математичній бібліотеці ***libm***. Отже, щоб підключити цю бібліотеку, ми повинні вказати опцію ***-lm***.

***$ gcc calculate.o main.o -o kalkul -lm***

***Компілятор G++***

Давайте знову переробимо нашу програму, але на цей раз вже напишемо її на **C++**. Створіть новий каталог проекту ***kalkulcpp***, а в ньому три файли: ***problem.h***, ***problem.cpp***, ***main.cpp***.

***problem.h***

***#ifndef PROBLEM\_H\_***

***#define PROBLEM\_H\_***

***#include <string>***

***using namespace std;***

***class CProblem***

***{***

***private:***

***float Numeral;***

***float SecondNumeral;***

***string Operation;***

***float Result;***

***string Error;***

***bool Calculate();***

***public:***

***void SetValues();***

***void Solve();***

***};***

***#endif /\*PROBLEM\_H\_\*/***

***problem.cpp***

***#include <iostream>***

***#include <cmath>***

***#include "problem.h"***

***using namespace std;***

***void CProblem::SetValues()***

***{***

***cout << "Число: ";***

***cin >> Numeral;***

***cout << "Арифметична дія (+,-,\*,/,pow,sqrt,sin,cos,tan): ";***

***cin >> Operation;***

***}***

***bool CProblem::Calculate()***

***{***

***if(Operation == "+")***

***{***

***cout << "Другий доданок: ";***

***cin >> SecondNumeral;***

***Result = Numeral + SecondNumeral;***

***return true;***

***}***

***else if(Operation == "-")***

***{***

***cout << "Від'ємник: ";***

***cin >> SecondNumeral;***

***Result = Numeral - SecondNumeral;***

***return true;***

***}***

***else if(Operation == "\*")***

***{***

***cout << "Добуток: ";***

***cin >> SecondNumeral;***

***Result = Numeral \* SecondNumeral;***

***return true;***

***}***

***else if(Operation == "/")***

***{***

***cout << "Дільник: ";***

***cin >> SecondNumeral;***

***if(SecondNumeral == 0)***

***{***

***Error = "Помилка: ділення на нуль.";***

***return false;***

***}***

***else***

***{***

***Result = Numeral/SecondNumeral;***

***return true;***

***}***

***}***

***else if(Operation == "pow")***

***{***

***cout << "Ступінь: ";***

***cin >> SecondNumeral;***

***Result = pow(Numeral,SecondNumeral);***

***return true;***

***}***

***else if(Operation == "sqrt")***

***{***

***Result = sqrt(Numeral);***

***return true;***

***}***

***else if(Operation == "sin")***

***{***

***Result = sin(Numeral);***

***return true;***

***}***

***else if(Operation == "cos")***

***{***

***Result = cos(Numeral);***

***return true;***

***}***

***else if(Operation == "tan")***

***{***

***Result = tan(Numeral);***

***return true;***

***}***

***else***

***{***

***Error = "Помилка введення дії.";***

***return false;***

***}***

***}***

***void CProblem::Solve()***

***{***

***if(Calculate() == true)***

***cout << Result << "\n";***

***else***

***cout << Error << "\n";***

***}***

***main.cpp***

***#include <iostream>***

***#include "problem.h"***

***using namespace std;***

***int main(void)***

***{***

***CProblem \*Problem;***

***Problem = new CProblem;***

***Problem->SetValues();***

***Problem->Solve();***

***delete Problem;***

***return(0);***

***}***

Для складання програм на **C++** в наборі **GNU** є інший компілятор - **G++**. Він відрізняється від **GCC** тим, що за замовчуванням підключати не стандартну бібліотеку C, а стандартну бібліотеку **C++**. Всі прапори і опції у **G++** такі ж точно, як і у **GCC**. Скористаємося компілятором **G++**.

***$ g++ problem.cpp main.cpp -o kalkul***

Зверніть увагу, що ніяких додаткових бібліотек ми не підключали. Це означає, що математичні функції входять в стандартну бібліотеку **C++**. Взагалі, в тих випадках, коли програма пишеться на **C++**, рекомендується використовувати саме **G++**.

Ви вже знаєте, що збирати програми таким простим способом можна тільки в навчальних цілях. У професійній же роботі ваші проекти будуть включати велику кількість файлів, і вам доведеться користуватися утилітою ***make***. Створимо ***Makefile***.

***kalkul: problem.o main.o***

***g++ problem.o main.o -o kalkul***

***problem.o: problem.cpp problem.h***

***g++ -c problem.cpp***

***main.o: main.cpp problem.h***

***g++ -c main.cpp***

***clean:***

***rm -f kalkul problem.o main.o***

***install:***

***cp kalkul /usr/local/bin/kalkul***

***uninstall:***

***rm -f /usr/local/bin/kalkul***

І зберемо цю ж програму знову, але вже «правильним» способом:

***$ make***

«Інсталюємо» її, попередньо зайшовши в систему, як привілейований користувач.

***$ su***

***$ make install***

***$ exit***

«Деінсталюємо».

***$ su***

***$ make uninstall***

***$ exit***

І очистимо дистрибутив.

***$ make clean***

**Лекція №14**

**«Управління файлами і каталогами у Linux програмним шляхом мовою С/С++»**

***Файли і їх метадані***

Кожен файл у Linux представляється структурою **inode** - індексного дескриптора, якій присвоюється унікальна в даній файлової системі числення величина, яка називається номером індексного дескриптора. Індексний дескриптор - і фізичний об'єкт, що знаходиться на диску файлової системи UNIX, і концептуальна сутність, яка надається структурою даних ядра Linux. Індексний дескриптор зберігає метадані, пов'язані з файлом, такі як права доступу до файлу, час останнього доступу, власник, група, розмір, а також розміщення даних файла.

Отримати номер індексного дескриптора можна за допомогою команди ***ls*** і прапора ***-i***:

***$ ls -i***

***1689459 Kconfig 1689461 main.c 1680144 process.c***

***1680137 Makefile 1680141 pm.c 1680145 smp.c***

***1680138 console.c 1689462 power.h 1689463 snapshot.c***

***1689460 disk.c 1680143 poweroff.c 1680147 swap.c***

Це виведення показує, що, наприклад, файл ***disk.c*** має номер індексного дескриптора ***1689460***. У цій конкретній файлової системи жоден інший файл не може мати такого ж номера. Про інші файлові системи, однак, ми не можемо стверджувати цього.

***Сімейство stat***

UNIX надає сімейство функцій для отримання метаданих файлу:

***#include <sys/types.h>***

***#include <sys/stat.h>***

***#include <unistd.h>***

***int stat(const char \*path, struct stat \*buf);***

***int fstat(int fd, struct stat \*buf);***

***int lstat(const char \*path, struct stat \*buf);***

Кожна з цих функцій повертає інформацію про файл.

Функція ***stat()*** надає інформацію про файл, визначеному через ***path***, в той час як ***fstat()*** повертає інформацію про файл, представлений файловим дескриптором ***fd***.

Функція ***lstat()*** ідентична ***stat()***, за винятком того, що в разі передачі їй символічного посилання ***lstat()*** повертає інформацію про саме посилання, а не про кінцевий файл.

Кожна з цих функцій зберігає інформацію в структурі ***stat***, яка надається користувачам. Структура ***stat*** визначена в ***<bits/stat.h>***, включеному в ***<sys/stat.h>***:

***struct stat***

***{***

***dev\_t st\_dev; /\*Ідентифікатор пристрою, на якому зберігається файл\*/***

***ino\_t st\_ino; /\*Номер індексного дескриптора\*/***

***mode\_t st\_mode; /\*Дозвіл\*/***

***nlink\_t st\_nlink; /\*Кількість жорстких посилань\*/***

***uid\_t st\_uid; /\*Призначений для користувача ідентифікатор власника\*/***

***gid\_t st\_gid; /\*Груповий ідентифікатор власника\*/***

***dev\_t st\_rdev; /\*Ідентифікатор пристрою (для спеціальних файлів\*/***

***off\_t st\_size; /\*Загальний розмір в байтах\*/***

***blksize\_t st\_blksize; /\*Розмір блоку для введення-виведення в файлової системі\*/***

***blkcnt\_t st\_blocks; /\*Кількість виділених блоків\*/***

***time\_t st\_atime; /\*Час останнього доступу\*/***

***time\_t st\_mtime; /\*Час останньої модифікації\*/***

***time\_t st\_ctime; /\*Час останньої зміни метаданих\*/***

***};***

Розглянемо поля докладніше.

* ***st\_dev*** – описує вузол пристрою, на якому зберігається файл (пізніше ми поговоримо про вузли пристроїв). Якщо файл не підтримується пристроєм, а знаходиться, наприклад, на ресурсі NFS, ця величина дорівнює 0;
* ***st\_ino*** – надає номер індексного дескриптора файлу.
* ***st\_mode*** – надає байти режиму файлу, які описують тип файлу (наприклад, це файл або директорія), а також привілеї доступу (наприклад, доступний всім тільки для читання).
* ***st\_nlink*** – призводить кількість жорстких посилань, пов'язаних з файлом. Кожен файл у файловій системі має хоча б одну жорстке посилання.
* ***st\_uid*** – призводить ідентифікатор користувача, який володіє файлом.
* ***st\_gid*** – призводить ідентифікатор групи, яка володіє файлом.
* Якщо файл є вузлом пристрою, то ***st\_rdev*** описує пристрій, який представляє цей файл.
* ***st\_size*** - представляє розмір файлу в байтах.
* ***st\_blksize*** – описує бажаний обсяг блоку, достатній для введення-виведення файлу. Ця величина (або кратне їй значення) - оптимальний розмір блоку для введення-виведення з користувацькою буферизацією.
* ***st\_blocks*** – надає кількість блоків в файлової системі, займаних файлом. Ця величина, помножена на розмір блоку, буде завжди менше, ніж значення, надане ***st\_size***, якщо у файлу є дірки (тобто якщо файл розріджений).
* ***st\_atime*** – містить час останнього доступу до файлу. Це найпізніший час, коли до файлу надавався доступ.
* ***st\_mtime*** – містить час останньої модифікації файлу, тобто час, коли в файл був зроблений останній запис.
* ***st\_ctime*** – містить час останньої зміни файлу. Поле містить час, коли метадані файлу (наприклад, його власник або права доступу) змінювалися в останній раз. Його часто плутають з часом створення файлу, який не зберігається в Linux та інших UNIX-подібних системах.

У разі успіху все три виклики повертають ***0*** і зберігають метадані файлу в структуру, надану ***stat***. При помилку вони повертають ***-1*** і привласнюють errno одне з наступних значень:

* ***EACCES*** – процес, який викликає, не має повноважень на пошук для одного з каталогів - компонентів шляху ***path*** (стосується тільки ***stat()*** і ***lstat()***);
* ***EBADF*** - значення ***fd*** є неприпустимим (стосується тільки ***fstat()***);
* ***EFAULT*** - покажчики ***path*** або ***buf*** є некоректними;
* ***ELOOP*** - шлях містить занадто багато символічних посилань (стосується тільки ***stat()*** і ***lstat()***);
* ***ENAMETOOLONG*** - ***path*** занадто великий (стосується тільки ***stat()*** і ***lstat()***);
* ***ENOENT*** - один з компонентів ***path*** не існує (стосується тільки ***stat()*** і ***lstat()***);
* ***ENOMEM*** - недостатньо пам'яті для виконання запиту;
* ***ENOTDIR*** - один з компонентів path не є директорією (стосується тільки ***stat()*** і ***lstat()***).

Наступна програма використовує ***stat()*** для отримання розміру файлу, вказаного в командному рядку:

***#include <sys/types.h>***

***#include <sys/stat.h>***

***#include <unistd.h>***

***#include <stdio.h>***

***int main (int argc, char \*argv[])***

***{***

***struct stat sb;***

***int ret;***

***if (argc < 2)***

***{***

***fprintf (stderr, "usage: %s <file>\n", arg);***

***return 1;***

***}***

***ret = stat(argv[1], &sb);***

***if (ret)***

***{***

***perror("stat");***

***return 1;***

***}***

***printf ("%s is %ld bytes\n", argv[1], sb.st\_size);***

***return 0;***

***}***

Запуск програми відносного її власного файлу призводить до наступного результату:

***$ ./stat stat.c***

***stat.c is 392 bytes***

***Дозволи***

За допомогою викликів stat можна отримати значення дозволів для даного файла, але для їх зміни використовуються два інших системних виклики:

***#include <sys/types.h>***

***#include <sys/stat.h>***

***int chmod(const char \* path, mode\_t mode);***

***int fchmod(int fd, mode\_t mode);***

Обидва виклику ***chmod()*** і ***fchmod()*** встановлюють ***mode*** в якості значення системних дозволів. У виклику ***chmod()*** відносний або абсолютний шлях до файлу вказується в змінної path. У ***fchmod()*** файл визначається файловим дескриптором ***fd***.

Допустимі величини ***mode***, представлені непрозорим цілочисельним типом ***mode\_t***, ті ж, що повертаються полем ***ds\_mode*** в структурі ***stat***. Значення належать до типу простих цілочисельних, проте їх значення специфічні для кожної реалізації **UNIX**. Отже, в **POSIX** визначається набір констант, що представляє все розмаїття змінних. Ці константи можуть бути об'єднані між собою за допомогою логічного «АБО», щоб сформувати допустимі значення mode. Наприклад, (***S\_IRUSR | S\_IRGRP***) встановлює значення, яке дозволяє читання файлу його власнику і групі.

Для зміни дозволів файлу дійсний ідентифікатор процесу, що викликає ***chmod()*** або ***fchmod()***, повинен відповідати власнику файлу або процес повинен мати властивість ***CAP\_FOWNER***.

Даний фрагмент коду встановлює файлу map.png дозволу на зчитування і запис для власника:

***int ret;***

***/\* Встановлення файлу 'map.png' в поточному каталозі дозволів на зчитування і запис для власника файлу. аналогічно команді 'Chmod 600 ./map.png'\*/***

***ret = chmod("./map.png", S\_IRUSR | S\_IWUSR);***

***if (ret)***

***perror("chmod");***

Даний фрагмент робить те ж, припускаючи, що ***fd*** представляє відкритий файл ***map.png***:

***int ret;***

***/\* Встановлення для файлу з дескриптором fd дозволів на читання і запис для власника файлу\*/***

***ret = fchmod(fd, S\_IRUSR | S\_IWUSR);***

***if (ret)***

***perror("fchmod");***

Обидва виклики, ***chmod()*** і ***fchmod()***, доступні у всіх сучасних системах **UNIX**. Стандарт **POSIX** вимагає реалізації першого, а другий вважає необов'язковим.

***Каталоги***

Концепція каталогів в **UNIX** дуже проста: вони містять список назв файлів, з кожним з яких пов'язаний номер індексного дескриптора. Кожне ім'я називаючи ється записом в каталозі, а кожна зв'язок імені і номера індексного дескриптора іменується посиланням. Вміст директорії, яке користувач бачить в результаті виконання команди ***ls***, є списком всіх імен файлів в цьому каталозі. Коли користувач відкриває файл в даному каталозі, ядро ​​шукає ім'я файлу в списку даної директорії, щоб визначити відповідний номер індекс ного дескриптора. Потім ядро ​​передає цей номер в файлову систему, яка використовує його, щоб визначити фізичне розміщення файлу на пристрої.

Назви файлів і папок можуть містити будь-які символи, крім ***/***, які поділяють каталоги в запису шляху до файлу, і порожнього значення (***null***), завершального шлях до файлу. Проте прийнято обмежувати використання символів в записах шляхи: як правило, використовуються тільки допустимі друковані символи з поточних мовних налаштувань або навіть тільки символи ASCII. Однак оскільки ні ядро, ні бібліотеки С не наказують таку практику, розробники додатків самостійно приймають рішення про допустимість використання тих чи інших символів.

У старих системах UNIX довжина імені файлу обмежувалася 14 символами. Сьогодні всі сучасні системи UNIX дозволяють використовувати для кожного файлу щонайменше 255 байт. (Зверніть увагу: мова йде не про 255 символах, а про 255 байтах. Багатобайтові символи, зрозуміло, займуть більше одного байта з цих 255.) Багато файлових систем під Linux дозволяють використання навіть довших імен файлів.

***Поточний робочий каталог***

Кожен процес має поточний каталог, який успадковує безпосередньо від свого батьківського процесу. Цей каталог називається поточним робочим каталогом процесу. Поточна робоча директорія є початковою точкою, з якою ядро ​​прокладає відносні шляхи до файлів. Наприклад, якщо поточна робоча директорія процесу ***/home/blackbeard***, то, коли процес намагається відкрити ***prrot.jpg***, ядро ​​буде намагатися відкрити файл ***/home/blackbeard/parrot.jpg***. Однак якщо процес спробує відкрити /usr/bin/mast, то ядро ​​справді відкриє ***/usr/bin/mast***. Поточний робочий каталог ніяк не впливає на абсолютні шляхи до файлів. Процес може отримати і змінити свою робочу папку.

***Отримання поточного робочого каталогу***

Переважний метод отримання поточного робочого каталогу – системний виклик ***getcwd()***, регламентований в **POSIX**:

***#include <unistd.h>***

***char \* getcwd(char \*buf, size\_t size);***

Успішний виклик ***getcwd()*** копіює поточний робочий каталог як абсолютний шлях в буфер, зазначений як ***buf*** і має довжину ***size*** байт, і повертає вказівник до ***buf***. У разі помилки виклик повертає ***NULL*** і привласнює errno одне з наступних значень:

* ***EFAULT*** - вказівник ***buf*** має неприпустиме значення;
* ***EINVAL*** - ***size*** дорівнює ***0***, але ***buf*** не дорівнює ***NULL***;
* ***ENOENT*** - поточний робочий каталог більш недійсний; це може статися, якщо поточний робочий каталог був видалений;
* ***ERANGE*** - ***size*** занадто малий, щоб поточний робочий каталог був збережений в ***buf***; додаток повинен виділити більший розмір буфера і спробувати знову.

Приклад використання ***getcwd()***:

***char cwd[BUF\_LEN];***

***if (!getcwd(cwd, BUF\_LEN))***

***{***

***perror("getcwd");***

***exit(EXIT\_FAILURE);***

***}***

***printf("cwd = %s\n", cwd);***

**POSIX** регламентує, що поведінка ***getcwd()*** не визначене, якщо ***buf*** дорівнює ***NULL***. Бібліотека **С** в Linux в даному випадку виділить буфер довжиною ***size*** байт і збереже там поточний робочий каталог. Якщо ***size*** дорівнює ***0***, бібліотека **С** виділить буфер достатнього розміру, щоб зберегти поточний робочий каталог. Потім додаток повинен очистити буфер через ***free()***, закінчивши роботу з ним. Ця поведінка унікальна для Linux, тому не можна покладатися на її надійну роботу в умовах сумісності. Проте ця функція робить використання дуже простим. Наприклад:

***char \*cwd;***

***cwd = getcwd(NULL, 0);***

***if (!cwd)***

***{***

***perror("getcwd");***

***exit(EXIT\_FAILURE);***

***}***

***printf("cwd =%s\n", cwd);***

***free(cwd);***

***Зміна поточного робочого каталогу***

Коли користувач вперше авторизується в системі, процес авторизації встановлює домашній каталог як поточний робочий, як зазначено в ***/etc/passwd***. Іноді, однак, процесу необхідно змінити свій поточний робочий каталог. Наприклад, оболонка може захотіти зробити це, коли користувач виконує команду ***cd***.

У Linux є два системні виклики для зміни поточного робочого каталогу: один, який встановлює шлях до каталогу, і інший, який прописує файловий дескриптор, що представляє відкритий каталог.

***#include <unistd.h>***

***int chdir(const char \*path);***

***int fchdir(int fd);***

Виклик ***chdir()*** змінює поточний робочий каталог згідно шляху, вказаному в ***path***, який може бути абсолютним або відносним. аналогічно виклик ***fchdir()*** змінює поточний робочий каталог згідно шляху, вказаному через файловий дескриптор ***fd***, який повинен бути відкритий для цього каталогу. В разі успіху обидва виклику повертають ***0***, при невдачі повертається ***-1***.

***Створення каталогів***

Для створення каталогів Linux надає тільки один системний виклик, регламентований **POSIX**:

***#include <sys/stat.h>***

***#include <sys/types.h>***

***int mkdir(const char \* path, mode\_t mode);***

Успішний виклик ***mkdir()*** створює шлях до каталогу ***path***, який може бути відносним або абсолютним з битами дозволу ***mode*** (як зазначено в поточному значенні ***umask***), і повертає ***0***.

Поточне значення ***umask*** змінює звичайним способом аргумент ***mode***, а також будь-які біти даного режиму, унікальні в цій системі. Іншими словами, параметр ***umask*** для даного процесу визначає значення, які не можуть бути перевизначені за допомогою виклику ***mkdir()***. Якщо для нового батьківського каталогу поточного каталогу встановлено загальний груповий ідентифікатор бітів (**set group ID**, або **sgid**) або файлова система змонтована з семантикою груп BSD, то новий каталог успадковує групову приналежність батька. В іншому випадку новий каталог отримає дійсний ідентифікатор групи процесу.

***Видалення каталогів***

На противагу ***mkdir()*** регламентований **POSIX** виклик ***rmdir()*** видаляє каталог з ієрархії файлової системи:

***#include <unistd.h>***

***int rmdir(const char \* path);***

У разі успіху ***rmdir()*** видаляє ***path*** з файлової системи і повертає ***0***. Каталог, вказаний через ***path***, повинен бути порожнім, за винятком вищезгаданих каталогів точка і точка-точка. Не існує системного виклику, що реалізує аналог рекурсивного видалення, на зразок ***rm -r***. Можна виконати подібну операцію вручну, просуваючись вглиб файлової системи і видаляючи всі файли і папки, починаючи з листя і переходячи до кореня файлової системи; на кожній стадії може бути використаний ***rmdir()*** для видалення каталогу відразу після того, як були видалені всі, хто знаходився в ньому файли.

Використання дуже просте:

***int ret;***

***/\* Видалення каталогу /home/barbary/ maps \*/***

***ret = rmdir("/home/barbary/maps");***

***if (ret)***

***perror ("rmdir");***

***Читання вмісту каталогу***

**POSIX** визначає сімейство функцій для читання вмісту каталогів, тобто для отримання списку файлів, які відносяться до даної директорії. Ці функції корисні, якщо ви реалізуєте ls або графічний інтерфейс діалогу збереження, виконання операцій з усіма файлами каталогу або пошук серед файлів каталогу тих, які відповідають певним шаблоном.

Щоб почати читання вмісту, ви повинні створити потік каталогу, який представлений об'єктом DIR:

***#include <sys/types.h>***

***#include <dirent.h>***

***DIR \* opendir(const char \* name);***

Успішний виклик ***opendir()*** створює потік каталогу, що представляє каталог, визначений через ***name***.

Потік каталогу є, по суті, просто файловим дескриптором, що представляє відкритий каталог, деякі метадані і буфер для збереження вмісту. Отже, можливо отримати дескриптор для відповідного каталогу всередині даного потоку каталогу:

***#define \_BSD\_SOURCE /\* або \_SVID\_SOURCE \*/***

***#include <sys/types.h>***

***#include <dirent.h>***

***int dirfd(DIR \* dir);***

Успішний виклик ***dirfd()*** повертає дескриптор, відповідний потоку каталогу ***dir***. У разі помилки виклик повертає ***-1***. Оскільки функція потоку каталогу використовує даний дескриптор при виконанні свого завдання, програми не повинні звертатися до викликів, що змінює положення файлів. ***dirfd()*** є розширенням BSD і не стандартизованй **POSIX**; програмісти, які прагнуть до повної відповідності стандартам **POSIX**, повинні уникати його.

***Читання з потоку каталогу***

Після того як за допомогою виклику ***opendir()*** створено потік каталогу, програма може читати знаходяться в ньому записи. Щоб зробити це, використовуйте ***readdir()***, який повертає записи одну за одною із зазначеного об'єкта ***DIR***:

***#include <sys/types.h>***

***#include <dirent.h>***

***struct dirent \* readdir(DIR \* dir);***

Успішний виклик ***readdir()*** повертає наступний запис в каталозі, представлену через ***dir***. Структура ***dirent*** представляє запис в каталозі. Вона визначена в заголовки ***<dirent.h>*** в Linux наступним чином:

***struct dirent***

***{***

***ino\_t d\_ino; /\* Номер inode \*/***

***off\_t d\_off; /\* Перехід до наступного запису dirent \* /***

***unsigned short d\_reclen; /\* Довжина даного запису \*/***

***unsigned char d\_type; /\* Тип файлу \*/***

***char d\_name [256]; /\* ім'я файлу \*/***

***};***

**POSIX** вимагає тільки наявності поля ***d\_name***, яке представляє собою ім'я файлу в даному каталозі. Інші поля необов'язкові або є унікальними для Linux. Додатки, що вимагають сумісності з іншими системами або суворого відповідності **POSIX**, повинні мати доступ тільки до ***d\_name***.

Додатки послідовно викликають ***readdir()***, отримуючи файли з каталогу один за іншим, поки записи в каталозі не закінчаться або виявлення нових файлів не припиниться. У першому випадку ***readdir()*** повертає значення ***NULL***.

У разі помилки ***readdir()*** також повертає ***NULL***. Щоб відрізнити помилку від закінчення списку файлів, перед кожним викликом ***readdir()*** слід привласнити змінної ***errno*** значення ***0***, а потім перевіряти значення, що підлягає поверненню, і ***errno***. Єдине значення ***errno***, яке може встановити ***readdir()***, - це ***EBADF***, яке вказує, що значення ***dir*** неприпустимо. Однак багато додатків не обробляють помилки, і в них передбачається, що повернення ***NULL*** означає лише закінчення файлів в каталозі.

***Закриття потоку каталогу***

Щоб закрити потік каталогу, відкритий за допомогою opendir(), потрібно використати функцію ***closedir()***:

***#include <sys/types.h>***

***#include <dirent.h>***

***int closedir(DIR \* dir);***

Успішний виклик ***closedir()*** закриває потік каталогу, позначеного за допомогою ***dir***, включаючи відповідний дескриптор, і повертає ***0***. В разі помилки функція повертає ***-1*** і встановлює ***errno*** в значення ***EBADF***, єдиний можливий код помилки, що означає, що dir не є відкритим потоком каталогу.

Наступний фрагмент коду представляє функцію ***find\_file\_in\_dir()***, яка використовує ***readdir()*** для пошуку даного імені файлу в зазначеному каталозі. Якщо файл дійсно знаходиться в каталозі, функція повертає ***0***. В іншому випадку вона повертає ненульову величину:

***int find\_file\_in\_dir(const char \* path, const char \* file)***

***{***

***struct dirent \* entry;***

***int ret = 1;***

***DIR \* dir;***

***dir = opendir(path);***

***errno = 0;***

***while ((entry = readdir(dir))!= NULL)***

***{***

***if (strcmp(entry->d\_name, file) == 0)***

***{***

***ret = 0;***

***break;***

***}***

***}***

***if (errno && !entry)***

***perror("readdir");***

***closedir(dir);***

***return ret;***

***}***

***Копіювання і переміщення файлів***

Дві найпростіші і поширені операції з файлами - це їх копіювання та переміщення, зазвичай виконуються за допомогою команд ***cp*** і ***mv***. На рівні файлової системи копіювання означає дублювання вмісту файлу в інше місце і створення нового шляху до файлу. Це відмінно від створення нової жорсткої посилання на файл, так як зміна одного файлу не впливає на інший: тепер існують дві різні копії файлу, яким відповідають дві (щонайменше) різні записи каталогу. Переміщення, навпаки, є актом перейменування запису каталогу, під яким розміщується файл. Ця дія не призводить до створення нової копії.

***Копіювання***

Напевно, це вас здивує, але в Linux немає системного або бібліотечного виклику для виконання копіювання файлів або каталогів. Замість цього утиліти на зразок cp або файлові менеджери GNOME виконують ці операції вручну.

Для копіювання файлу ***src*** в файл під назвою ***dst*** необхідно виконати наступні дії:

1. Відкрити файл ***src***;

2. Відкрити файл ***dst*** - створити його, якщо він не існує, або скоротити до нульової довжини, якщо він є;

3. Вважати фрагмент вмісту ***src*** в пам'ять;

4. Записати цей фрагмент в ***dst***;

5. Продовжувати, поки весь вміст src не буде зчитаний і переписаний в ***dst***;

6. Закрити ***src***;

7. Закрити ***dst***.

При копіюванні каталогу окремий каталог і його підкаталоги створюються за допомогою ***mkdir()***; кожен файл всередині них потім копіюється індивідуально.

***Переміщення***

На відміну від копіювання файлів, UNIX надає системний виклик для переміщення. Стандарт ANSI C являє виклик для файлів, а **POSIX** регламентує виклик і для файлів, і для каталогів:

***#include <stdio.h>***

***int rename(const char \* oldpath, const char \* newpath);***

Успішний виклик ***rename()*** перейменовує шлях ***oldpath*** в ***newpath***. Вміст файлу і ***inode*** залишаються тими ж самими. І ***oldpath***, і ***newpath*** повинні залишитися в тій же файлової сістемі; якщо це не так, виклик поверне помилку. Утиліти на зразок ***mv*** обробляють цей випадок копіюванням файлу і видаленням посилання.

При успіху ***rename()*** повертає ***0***, і шлях до файлу ***oldpath*** змінюється на ***newpath***. У разі невдачі виклик повертає ***-1***, ніяк не впливаючи ні на ***oldpath***, ні на ***newpath***, і привласнює ***errno*** одне з наступних значень:

* ***EACCES*** - викликає, не має прав запису в батьківських каталогах ***oldpath*** або ***newpath***, прав пошуку в одному з компонентів ***oldpath*** або ***newpath*** або прав запису для ***oldpath***, якщо це каталог; останній випадок може представляти проблему, так як ***rename()*** повинен оновити в ***oldpath***, якщо це каталог;
* ***EBUSY*** - ***oldpath*** або ***newpath*** є точкою монтування;
* ***EFAULT*** - вказівник ***oldpath*** або ***newpath*** є неприпустимим;
* ***EINVAL*** - ***newpath*** знаходиться всередині ***oldpath***; таким чином, перейменування одного в інший зробить ***oldpath*** підкаталогом самого себе;
* ***EISDIR*** - ***newpath*** існує і є каталогом, а ***oldpath*** не є;
* ***ELOOP*** - при вирішенні шляху ***oldpath*** або ***newpath*** було виявлено занадто багато символічних посилань;
* ***EMLINK*** - ***oldpath*** вже досяг максимально допустимої кількості посилань на себе або ***oldpath*** є каталогом, а ***newpath*** вже досяг максимально допустимої кількості посилань на себе;
* ***ENAMETOOLONG*** - шляхи ***oldpath*** або ***newpath*** занадто великі;
* ***ENOENT*** - один з компонентів ***oldpath*** або ***newpath*** не існує або є повисле символічне посилання;
* ***ENOMEM*** - недостатньо пам'яті ядра для виконання запиту;
* ***ENOSPC*** - на пристрої, де знаходиться newpath, недостатньо простору для виконання запиту;
* ***ENOTDIR*** - один з компонентів (за винятком потенційно останнього компонента) ***oldpath*** або ***newpath*** не є каталогом або ***oldpath*** - це каталог, а ***newpath*** вже існує і не є каталогом;
* ***ENOTEMPTY*** - ***newpath*** є каталогом і непорожній;
* ***EPERM*** - принаймні один із зазначених шляхів існує, для батьківського каталогу встановлений біт закріплення в пам'яті, дійсний ідентифікатор користувача викликає процесу не збігається ні з ідентифікатором користувача файлу, ні з ідентифікатором користувача батьківського процесу, а процес не має привілейованих прав;
* ***EROFS*** - файлова система доступна тільки для читання;
* ***EXDEV*** - ***oldpath*** і ***newpath*** знаходяться в різних файлових системах.

У табл. 14.1 наведені результати переміщення різних типів файлів.

Табл. 14.1. результати переміщення різних типів файлів

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Вихідний об'єкт є** | **Мета - файл** | **Мета - каталог** | **Мета - посилання** | **Мета не існує** |
| файлом | Цільовий файл замінюється на вихідний | Помилка ***EISDIR*** | Файл перейменовується, мета переписується | Файл перейменовується |
| каталогом | Помилка ***ENOTDIR*** | Якщо цільовий каталог порожній, то вихідний файл перейменовується назвою цільового файлу; в іншому випадку повертається помилка ***ENOTEMPTY*** | Каталог перейменовується,  мета переписується | Каталог перейменовується |
| посиланням | Посилання перейменовується, мета переписується | Помилка ***EISDIR*** | Посилання перейменовується, мета переписується | Посилання перейменовується |
| Не існує | Помилка ***ENOENT*** | Помилка ***ENOENT*** | Помилка ***ENOENT*** | Помилка ***ENOENT*** |

У всіх випадках, незалежно від типів вихідного і цільового об'єктів, якщо вони знаходяться в різних файлових системах, виклик поверне помилку ***EXDEV***.

**Лекція №15**

**«Управління процесами в Linux мовами C та C++. Системні виклики exec() та fork(). Завершення процесів»**

**Процеси** - це одна з найбільш фундаментальних абстракцій в системах UNIX після файлів. Якщо розглядати процеси як об'єкт виконуваного коду - живий, що працює, активної програми, - вони є чимось більшим, ніж просто мова асемблера, і складаються з даних, ресурсів, стану і віртуального процесора.

У даній лекції ми розглянемо основні компоненти процесу, від створення до завершення. Основи не зазнали значних змін з перших днів існування UNIX. Саме тут, в управлінні процесами, найяскравіше проявляються далекоглядність і довговічність початкової концепції UNIX. Історія системи цікава і вельми неабияка, так як в концепції UNIX створення нового процесу відділяється від завантаження нового довічного образу. Хоча в більшості випадків ці дії виконуються спільно, поділ дозволяє експериментувати і відкриває широкі можливості для розвитку кожного завдання. Таким чином, ця концепція зберіглася до наших днів, в той час як більшість операційних систем пропонують один системний виклик для запуску однієї програми, в UNIX потрібні два: ***fork*** і ***ехес***. Однак до вивчення цих системних викликів давайте детально розглянемо сам процес.

***Програми, процеси і потоки***

**Бінарний модуль** - це виконуваний код, який компілюється та знаходиться в якому-небудь сховищі даних, наприклад на диску. Отже, ми можемо також використовувати термін «**програма**»; великі і значні бінарні модулі ми можемо називати **додатками**. І ***/bin/ls***, і ***/usr/bin/X11*** є бінарними модулями.

**Процесом** є запущена програма. Процес включає в себе бінарний образ, що завантажується в пам'ять, і багато іншого: підзавантаження віртуальної пам'яті, ресурси ядра, наприклад відкриті файли, виконання вимог з безпеки (Наприклад, вибір певного користувача), а також запуск одного або декількох потоків. **Потік** - це одне з дій всередині процесу. кожен потік має власний віртуалізувати процесор, що включає в себе стек, стан процесора, наприклад регістри, а також командні вказівники.

Якщо в процесі потік тільки один, то процес і є потоком. У нього тільки один екземпляр віртуальної пам'яті і один віртуалізований процесор. В багатопоточних процесах потоків кілька. Віртуалізація пам'яті пов'язана з процесом, тому всі потоки одночасно використовують один і той же адресний простір пам'яті.

***Ідентифікатор процесу***

Кожен процес позначається унікальним ідентифікатором (**process ID**, зазвичай скорочується до **PID**). **PID** обов'язково є унікальним в будь-який конкретний момент часу. Це означає, що в момент часу t+0 може бути тільки один процес з ***pid 770*** (або жодного процесу з таким значенням ідентифікатора), але не можна гарантувати, що в момент часу t+1 не існуватиме іншого процесу з тим же ідентифікатором ***pid 770***. На практиці, втім, ядро ​​не обов'язково швидко міняє місцями ідентифікатори процесів - припущення, що є, як ви скоро побачите, вельми небезпечним. Звичайно, з точки зору самого процесу, його ***pid*** залишається незмінним.

Процес бездіяльності, або пасивний, який виконується ядром за відсутності всіх інших процесів, має ***pid 0***. Перший процес, який ядро ​​виконує під час запуску системи, називається процесом ініціалізації і має ***pid 1***. Зазвичай **init process** в Linux є програмою ініціалізації. Ми використовуємо термін «ініціалізація» для позначення і початкового процесу, що запускається при завантаженні, і спеціальної програми, яка використовується для цих цілей.

***Виділення ідентифікатора процесу***

За замовчуванням ядро ​​може виставити максимальну величину ідентифікатора, рівну **32768**. Це пояснюється необхідністю сумісності з більш старими системами UNIX, які використовували 16-бітові типи даних для ідентифікаторів. Адміністратор системи може встановити більшу величину через ***/proc/sys/kernel/pid\_max***, виділивши більший простір для pid, але знизивши таким чином сумісність.

Ідентифікатори призначаються ядром строго лінійно. Якщо в даний момент найбільший наявний ***pid*** дорівнює ***17***, то наступною буде призначена величина ***18***, навіть якщо процес з останнім призначеним ідентифікатором ***17*** вже не виконується під час старту нового процесу. Ядро не призначить використані раніше ідентифікатори процесів, поки не пройде верхнє значення, тобто менші значення не будуть встановлюватися, поки не буде досягнута величина, записана в ***/proc/sys/kernel/pid\_max***. Таким чином, Linux не гарантує унікальність ідентифікаторів процесів на довгі періоди, але тим не менш є деяка впевненість, що протягом коротких відрізків часу ідентифікатори будуть стабільними і унікальними.

***Ієрархія процесів***

Процес, що запускає інший процес, називається **батьківським**; новий процес, таким чином, є **дочірнім**. Кожен процес запускається будь-яким іншим способом (крім, зрозуміло, процесів ініціалізації). Таким чином, кожен дочірній процес має «батька». Ці взаємини записані в кожному **ідентифікаторі батьківського процесу (ppid)**, значення якого для дочірнього процесу дорівнює значенню **pid** батьківського процесу.

Кожен процес належить певному користувачеві і групі. Ці приналежності використовуються для управління правами доступу до ресурсів. З точки зору ядра користувач і група - це просто якісь цілочисельні величини. Вони зберігаються в файлах ***/etc/passwd*** і ***/etc/group***, за допомогою яких зіставляються з звичними оку користувача UNIX іменами, які сприймалися людиною, такі як імена користувачів **root** або група **wheel** (в загальному випадку ядро ​​Linux ніяк не взаємодіє з цими рядками, оперуючи з об'єктами за допомогою цілочисельних величин). Кожен дочірній процес успадковує користувача і групу, яким належав батьківський процес.

Кожен процес є також частиною групи процесів, яка означає, по суті, його відношення до інших. Дочірні процеси, як правило, належать до тих же групам процесів, що і батьківські. Крім того, коли користувач запускає конвеєр (наприклад, ввівши ***ls | less***), всі команди в конвеєрі стають членами однієї і тієї ж групи процесів. Поняття групи процесів спрощує відправку сигналів або отримання інформації від усього конвеєра, так як всі дочірні процеси знаходяться в конвеєрі. З точки зору користувача група процесів тісно пов'язана з поняттям завдання.

З точки зору програмування ідентифікатор процесу позначається типом ***pid\_t***, величина якого визначається в заголовки ***<sys/types.h>***. Конкретний тип **С** залежить від архітектури і не визначається будь-яким стандартом **С**. В Linux, однак, для ***pid\_t*** найчастіше використовується тип даних **С** ***int***.

***Отримання ідентифікаторів процесу і батьківського процесу***

Системний виклик ***getpid()*** повертає ідентифікатор процесу:

***#include <sys/types.h>***

***#include <unistd.h>***

***pid\_t getpid(void);***

Системний виклик ***getppid()*** повертає ідентифікатор батька викликає процесу:

***#include <sys/types.h>***

***#include <unistd.h>***

***pid\_t getppid (void);***

Жоден з них не може повернути помилку, тому використання даних викликів нескладно:

***printf("My pid =%jd\n", (intmax\_t) getpid());***

***printf("Parent's pid =%jd\n", (intmax\_t) getppid());***

Тут ми наводимо повернуту величину до типу ***intmax\_t***, який є типом **С/С++**, гарантовано здатним зберігати в системі будь-яке ціле число зі знаком. Іншими словами, він дорівнює будь-яким іншим типам для цілих чисел зі знаками або більше їх. У поєднанні з ***printf()*** з модифікатором (***%j***) такий підхід дозволяє нам безпечно друкувати змінні типу ***integer***, призначені ***typedef***. До появи ***intmax***\_t не існувало прийнятного способу зробити це (якщо у вашій системі немає ***intmax\_t***, ви можете припустити, що ***pid\_t*** належить до ***int***, що правдиво для більшості систем UNIX).

***Запуск нового процесу***

В UNIX дію завантаження в пам'ять і запуску образу програми виконується окремо від операції по створенню нового процесу. Один системний виклик завантажує бінарну програму в пам'ять, заміщаючи поточне утримання адресного простору, і починає виконання нової програми. Це називається виконанням нової програми, а функціональність забезпечується сімейством викликів ***exec***.

Інший системний виклик використовується для створення нового процесу, який спочатку є практично копією свого батьківського. Часто новий процес негайно приступає до виконання нової програми. Акт створення нового процесу називається розгалуженням і забезпечується системним викликом ***fork()***. Дві дії - спочатку розгалуження для створення нового процесу, а потім ***exec*** для відкриття нового виконання цього процесу - потрібні для запуску нової програми в новому процесі. Вивчимо виклики ***exec*** і ***fork()***.

***Сімейство викликів exec***

Єдиної функції ***exec*** не існує; на одному системному виклику побудовано ціле сімейство таких функцій. Спочатку розглянемо найпростіший з цих викликів, ***execl()***:

***#include <unistd.h>***

***int execl (const char \* path, const char \* arg, ...);***

Виклик ***execl()*** заміщає поточний образ процесу новим, завантажуючи в пам'ять програму, визначену ***path***. Параметр ***arg*** - перший аргумент цієї програми. Три крапки означає змінну кількість аргументів - у функції ***execl()*** їх кількість може бути будь-яким, додаткові аргументи можна вказувати в дужках один за іншим. Список аргументів завжди завершується значенням ***NULL***.

Наприклад, наступний програмний код заміщає виконувати зараз програму з ***/bin/vi***:

***int ret;***

***ret = execl ("/bin/vi", "vi", NULL);***

***if (ret == -1)***

***perror("execl");***

Зверніть увагу: слідуючи угодами UNIX, ми передаємо в якості першого аргументу програми значення ***"vi"***. Оболонка поміщає останній компонент шляху, тобто ***"vi"***, в перший аргумент під час розгалуження або запуску процесів, завдяки чому програма може перевірити перший аргумент ***argv[0]*** для з'ясування імені довічного образу. У більшості випадків кілька системних утиліт, що відображаються користувачу під різними іменами, в дійсності є єдиною програму з жорстко прописаними посиланнями до різних іменах. Програма використовує перший аргумент, щоб визначити свою поведінку.

Інший приклад. Якщо ви хочете редагувати файл ***/home/kidd/hooks.txt***, то повинні запустити наступний код:

***int ret;***

***ret = execl("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);***

***if (ret == -1)***

***perror ( "execl");***

Як правило, ***execl()*** не повертає ніяких значень. Успішний виклик завершується переходом до вхідних точці нової програми, і тільки що виконаний код більше не знаходиться в процесному адресному просторі. Якщо сталася помилка, ***execl()*** повертає ***-1*** і встановлює ***errno*** для позначення проблеми. Ми розглянемо можливі значення ***errno*** в цьому розділі пізніше.

У разі успішного виконання виклик ***execl()*** змінює не тільки адресний простір і образ процесу, а й деякі інші атрибути процесу:

* будь-які сигнали, що очікують, зникають;
* будь-які сигнали, які відловлюють процесом, повертаються до своєї поведінки за замовчуванням, оскільки оброблювачів сигналів більше немає в процесному адресному просторі;
* всі блокування пам'яті видаляються;
* більшість атрибутів потоку повертається до значень за замовчуванням;
* велика частина статистичних даних процесу скидається;
* весь адресний простір пам'яті, що відноситься до даного процесу, включаючи завантажені файли, очищається;
* все, що знаходиться виключно в просторі користувача, включаючи функціональності бібліотеки **С**, наприклад поведінка ***atexit()***, видаляється.

Деякі властивості процесу, проте, не змінюються. Наприклад, ідентифікатор (свій і батьківський), пріоритет, а також користувач та група залишаються тими ж.

Зазвичай при роботі системних викликів сімейства ***exec*** успадковуються і відкриті файли: у запущених програм зберігається повний доступ до всіх файлів, відкритим в первісному процесі, якщо їм відомі значення дескрипторів. Однак найчастіше це небажано. Звичайною практикою є закриття файлів перед запуском ***exec***, але можливо дати команду ядру робити це автоматично через ***fcntl()***.

***Інша частина сімейства***

Крім ***execl()***, в сімействі є ще п'ять членів:

***#include <unistd.h>***

***int execlp (const char \* file, const char \* arg, ...);***

***int execle (const char \* path, const char \* arg, ..., char \* const envp []);***

***int execv (const char \* path, char \* const argv []);***

***int execvp (const char \* file, char \* const argv []);***

***int execve (const char \* filename,***

***char \* const argv [],***

***char \* const envp []);***

Запам'ятати все дуже просто. ***l*** і ***v*** вказують, чи передаються аргументи списком або масивом. Символ ***p*** вказує, що система буде шукати вказаний файл за повним призначеному для користувача шляху. У командах, де використовуються варіанти з ***p***, можна вказати тільки ім'я файлу, якщо він знаходиться в межах призначеного для користувача шляху. Нарешті, е позначає, що для нового процесу створюється нове оточення. Цікаво, що, хоча технічних обмежень для цього не існує, в сімействі ***exec*** немає елемента, що дозволяє і шукати шлях до файлів, і створювати нове оточення. Можливо, це пояснюється тим, що варіанти p призначені для використання оболонками, а процеси, що виконуються в оболонках, як правило, успадковують своє оточення від них.

У наступному фрагменті коду використовується ***execvp()*** для виконання ***vi***, як і в попередньому варіанті, враховуючи, що ***vi*** знаходиться на призначеному для користувача шляху:

***int ret;***

***ret = execvp ("vi", "vi", "/home/kidd/hooks.txt", NULL);***

***if (ret == -1)***

***perror("execvp");***

Елементи сімейства ***exec***, які беруть в якості аргументу масив, працюють точно так само, за одним винятком - замість списку генерується і передається масив. Використання масиву дозволяє визначати аргументи під час виконання програми. Як і в разі аргументу у вигляді списку змінної довжини, масив повинен закінчуватися значенням ***NULL***.

У наступному прикладі ***execv()*** використовується для запуску vi, як ми вже робили раніше:

***const char \* args [] = {"vi", "/home/kidd/hooks.txt", NULL};***

***int ret;***

***ret = execv ("/bin/vi", args);***

***if (ret == -1)***

***perror ( "execvp");***

У Linux тільки один елемент сімейства ***exec*** є системним викликом. Решта - це оболонки системного виклику в бібліотеці ***С***. Системні виклики зі змінною кількістю аргументів складні в реалізації, а концепція призначеного для користувача шляху існує тільки в просторі користувача, єдиним елементом, здатним здійснювати функції системного виклику, є ***execve()***. Прототип системного виклику ідентичний призначеному для користувача виклику.

***Системні виклики fork()***

Новий процес, який запускає той же системний образ, що і поточний, може бути створений за допомогою системного виклику ***fork()***:

***#include <sys/types.h>***

***#include <unistd.h>***

***pid\_t fork(void);***

У разі успішного звернення до ***fork()*** створюється новий процес, в усіх відношеннях ідентичний тому, що викликає. Обидва процеси виконуються від точки звернення до ***fork()***, як ніби нічого не відбувалося.

Новий процес є **дочірнім** по відношенню до того, який викликає, який, в свою чергу, називається **батьківським**. У дочірньому процесі успішний запуск ***fork()*** повертає ***0***. В батьківському ***fork()*** повертає ***pid*** дочірнього. Батьківський і дочірній процеси практично ідентичні, за винятком деяких особливостей:

* ***pid*** дочірнього процесу, звичайно, призначається заново і відрізняється від батьківського;
* батьківський ***pid*** дочірнього процесу встановлено рівним ***pid*** батьківського процесу;
* ресурсна статистика дочірнього процесу обнуляється;
* будь-які сигнали, що очікують, перериваються і не успадковуються дочірнім процесом;
* ніякі залучені блокування файлів не успадковуються дочірнім процесом.

У разі помилки дочірній процес не створюється, ***fork()*** повертає ***-1***, встановлюючи відповідне значення ***errno***. Ось два можливих значення ***errno*** і їх зміст:

* ***EAGAIN*** - ядро ​​не здатне виділити певні ресурси, наприклад новий ***pid***, або досягнуте обмеження за ресурсами ***RLIMIT\_NPROC***;
* ***ENOMEM*** - недостатньо ресурсів пам'яті ядра, щоб завершити запит.

Використання дуже просте:

***pid\_t pid;***

***pid = fork();***

***if (pid> 0)***

***printf("Я батьківський процес сpid =%d!\n", pid);***

***else if (!pid)***

***printf("А я дочірній!\n");***

***else if (pid == -1)***

***perror("fork");***

Найчастіше системний виклик ***fork()*** використовується для створення нового процесу і подальшого завантаження в нього нового довічного образу. Уявімо собі оболонку, в якій користувач запускає новий додаток або процес починає допоміжну програму. Спочатку процес відокремлює новий процес, а потім дочірній процес створює новий двійковий образ. Поєднання ***fork*** і ***exec*** використовується часто і без ускладнень. У наступному прикладі відгалужується новий процес, який запускає бінарний файл ***/bin/windlass***:

***pid\_t pid;***

***pid = fork();***

***if (pid == -1)***

***perror("fork");***

***/\* Дочірній ... \*/***

***if (!pid)***

***{***

***const char \*args[] = {"windlass", NULL};***

***int ret;***

***ret = execv("/bin/windlass", args);***

***if (ret == -1)***

***{***

***perror("execv");***

***exit(EXIT\_FAILURE);***

***}***

***}***

Батьківський процес залишиться активним, як і раніше, за винятком появи у нього нового дочірнього. Виклик ***execv()*** впливає тільки на дочірній процес, змушуючи його виконати програму ***/bin/windlass***.

***Завершення процесу***

**POSIX** і **C89** визначають наступну стандартну функцію для завершення поточного процесу:

***#include <stdlib.h>***

***void exit(int status);***

Виклик ***exit()*** виконує деякі основні кроки перед завершенням, а потім відправляє ядру команду припинити процес. Ця функція не може повернути помилку - по суті, вона взагалі не повертає ніяких результатів. Отже, немає сенсу в будь-яких інших інструкціях щодо виконання виклику ***exit()***.

Параметр ***status*** використовується для позначення статусу процесу завершення. Інші програми - як і користувач оболонки - можуть перевіряти цю величину. Значення ***EXIT\_SUCCESS*** і ***EXIT\_FAILURE*** визначаються як способів подання успіху і невдачі. У Linux значення ***0*** зазвичай означає успіх, а будь-який ненульове значення, наприклад ***-1*** або ***1***, відповідає невдачі.

Отже, успішний вихід - єдиний рядок:

***exit (EXIT\_SUCCESS);***

Перед тим як перервати процес, бібліотека **C** виконує підготовчі кроки в наступному порядку.

1. Виклик всіх функцій, зареєстрованих з ***atexit()*** або ***on\_exit()***, в порядку, зворотному порядку реєстрації;
2. Скидання всіх стандартних потоків введення-виведення;
3. Видалення всіх тимчасових файлів, створених функцією ***tmpfile()***.

Ці кроки завершують всю роботу, яку процес повинен виконати в просторі користувача, після чого ***exit()*** виконує системний виклик ***\_exit()***, що дозволяє ядру обробити решту завершення процесу:

***#include <unistd.h>***

***void \_exit(int status);***

Коли процес завершується, ядро ​​очищає всі ресурси, які були виділені для потреб процесу і більш не використовуються. До них відносяться виділена пам'ять, відкриті файли, семафори System V і ін. Після очищення ядро ​​знищує процес і попереджає батьківський процес про завершення дочірнього.

Додатки можуть викликати ***\_exit()*** безпосередньо, але найчастіше це позбавлене сенсу: більшості додатків необхідно провести деяку зачистку перед повним виходом, наприклад перервати потік ***stdout***. Зверніть увагу, однак, що при використанні ***vfork()*** після розгалуження потрібно викликати ***\_exit()***, а не ***exit()***.

Прагнучи до надмірної точності, стандарт ISO C99 додав функцію ***\_Exit()***, поведінка якої повністю ідентична ***\_exit()***:

***#include <stdlib.h>***

***void \_Exit(int status);***

***Очікування завершених дочірніх процесів***

Отримання попередження на сигнал - це прекрасно, але багато батьківськіх процесів хочуть отримати більше інформації, коли завершується їх дочірній процес - наприклад, якщо той повертає якесь значення.

Якби нащадок по завершенні повністю зникав, як можна було б припустити, то отримання будь-яких відомостей було б для його пращура неможливо. Отже, перші розробники UNIX вирішили, що коли дочірній процес завершується перш батьківського, ядро ​​має помістити нащадка в особливий процесний статус. Процес в цьому стані відомий як зомбі. В даному стані існує лише «скелет» процесу - деякі основні структури даних, що містять потенційно потрібні відомості. Процес в такому стані очікує запиту про свій статус від пращура (процедура, відома як очікування процесу-зомбі). Тільки після того як пращур отримає всю необхідну інформацію про завершеному дочірньому процесі, останній формально видаляється і перестає існувати навіть в статусі зомбі.

Ядро Linux надає кілька інтерфейсів для отримання інформації про завершеному дочірньому процесі. Найпростіший з них, визначений в **POSIX**, називається ***wait()***:

***#include <sys/types.h>***

***#include <sys/wait.h>***

***pid\_t wait(int \*status);***

Виклик ***wait()*** повертає ***pid*** завершеного дочірнього процесу або ***-1*** у випадку помилки. Якщо ніякого дочірнього процесу не було перервано, виклик блокується, поки нащадок не завершиться. Якщо дочірній процес уже був завершений, виклик повертає результати негайно. Отже, якщо викликати ***wait()*** відразу після повідомлення про завершення дочірнього процесу, результат буде негайно видано без блокування.

У разі помилки можливо привласнення змінної ***errno*** одного з двох значень:

* ***ECHILD*** – процес, що викликає, не має дочірніх;
* ***EINTR*** - сигнал був отриманий під час очікування, в результаті чого виклик повернув результат занадто рано.

Якщо покажчик ***status*** не містить значення ***NULL***, там знаходиться додаткова інформація про дочірньому процесі. **POSIX** дозволяє при реалізації визначення бітів статусу розробниками самостійно, тому стандарт передбачає сімейство макросів для інтерпретації параметра:

***#include <sys/wait.h>***

***int WIFEXITED(status);***

***int WIFSIGNALED(status);***

***int WIFSTOPPED(status);***

***int WIFCONTINUED(status);***

***int WEXITSTATUS(status);***

***int WTERMSIG(status);***

***int WSTOPSIG(status);***

***int WCOREDUMP(status);***

Будь-який з перших двох макросів може повертати значення ***true*** (нульове) в залежності від ходу завершення процесу. Перший, ***WIFEXITED***, повертає ***true***, якщо процес завершується через виклик ***\_exit()***, звичайним чином. Відповідно, макрос ***WEXITSTATUS*** надає 8 біт молодших розрядів і передає їх ***\_exit()***.

Макрос ***WIFSIGNALED*** повертає ***true***, якщо переривання процесу викликав сигнал. В цьому випадку ***WTERMSIG*** повертає номер сигналу, який викликав переривання, а ***WCOREDUMP*** повертає ***true***, якщо процес скинув ядро ​​у відповідь на отримання сигналу. ***WCOREDUMP*** не визначається **POSIX**, хоча багато систем UNIX, Linux в тому числі, підтримують його.

Макроси ***WIFSTOPPED*** і ***WIFCONTINUED*** повертають ***true***, якщо процес був зупинений або продовжений відповідно і його можна зараз відстежити за допомогою системного виклику ***ptrace()***. Ці умови зазвичай можливі тільки під час реалізацій відладчика, хоча при використанні разом з ***waitpid()***вони можуть використовуватися для контролю роботи. Зазвичай ***wait()*** застосовується тільки для обміну інформацією про завершення процесу. Якщо ***WIFSTOPPED*** повернув ***true***, ***WSTOPSIG*** призводить номер сигналу, який зупинив процес. ***WIFCONTINUED*** не визначається **POSIX**, хоча більш пізні стандарти визначили його для ***waitpid()***. У версії 2.6.10 ядра Linux також надано макрос для ***wait()***.

Розглянемо приклад програми, яка використовує ***wait()***, щоб визначити, що сталося з дочірнім процесом:

***#include <unistd.h>***

***#include <stdio.h>***

***#include <sys / types.h>***

***#include <sys / wait.h>***

***int main (void)***

***{***

***int status;***

***pid\_t pid;***

***if (!fork())***

***return 1;***

***pid = wait (&status);***

***if (pid == -1)***

***perror ("wait");***

***printf ("pid =%d\n", pid);***

***if (WIFEXITED(status))***

***printf ("Нормальне завершення, статус =%d\n", WEXITSTATUS (status));***

***if (WIFSIGNALED (status))***

***printf ("Вбитий сигналом =%d%s\n", WTERMSIG (status), WCOREDUMP (status)? "(Dumped core)": "");***

***if (WIFSTOPPED (status))***

***printf ("Зупинено сигналом =%d\n", WSTOPSIG (status));***

***if (WIFCONTINUED (status))***

***printf ("Продовжено \ n");***

***return 0;***

***}***

Ця програма відокремлює дочірній процес, який негайно завершується. Після цього предок запускає системний виклик ***wait()*** для визначення статусу нащадка. Процес друкує ***pid*** нащадка і відомості про його завершення. Оскільки в цьому випадку дочірній процес завершився поверненням результату ***main()***, зрозуміло, що на виході ми отримаємо приблизно наступне:

***$ ./wait***

***pid = 8529***

***Нормальне завершення зі статусом виходу = 1***

Якщо замість звичайного повернення дочірнього процесу реалізований виклик ***abort()***, який відправляє процесу сигнал ***SIGABRT***, то результат буде виглядати так:

***$ ./wait***

***pid = 8678***

***Перерваний сигналом = 6***

***Очікування певного процесу***

Спостереження за поведінкою дочірнього процесу дуже важливо. Часто, однак, процес має кількох нащадків і, якщо потрібен тільки певний, очікування всіх небажано. Можливим рішенням були б багаторазові виклики ***wait()*** з постійною перевіркою значення, що повертається, проте це дуже незручно: що, якщо пізніше знадобиться перевірити статус іншого завершеного процесу? Батьківському процесу довелося б зберігати результати всіх викликів ***wait()*** на випадок, якщо вони знадобляться пізніше.

Якщо вам відомий ***pid*** процесу, завершення якого ви чекаєте, можна використовувати системний виклик ***waitpid()***:

***#include <sys/types.h>***

***#include <sys/wait.h>***

***pid\_t waitpid(pid\_t pid, int \* status, int options);***

Виклик ***waitpid()*** - більш потужна версія ***wait()***. Його додаткові параметри дозволяють налаштувати його більш тонко.

Параметр ***pid*** точно визначає, який процес або процеси потрібно очікувати. Його значення можуть потрапляти в чотири проміжки:

* ***<-1*** - очікування будь-якого дочірнього процесу, чий ID групи процесів дорівнює абсолютному значенню цієї величини; наприклад, при ***-500*** очікується будь-який процес з групи процесів ***500***;
* ***-1*** - очікування будь-якого дочірнього процесу; поведінка аналогічна ***wait()***;
* ***0*** - очікування будь-якого дочірнього процесу, що належить тій же групі процесів, що і той, який викликає;
* ***>0*** - очікування будь-якого дочірнього процесу, чий ***pid*** в точності дорівнює зазначеної величині; наприклад, при величині ***500*** очікується дочірній процес з ***pid***, рівним ***500***.

Параметр ***status*** працює аналогічно такому в системному виклику ***wait()*** і може бути оброблений за допомогою макросів, описаних вище.

Параметр ***options*** може передавати наступні значення за допомогою логічного «АБО» або пусте значення:

* ***WNOHANG*** - не блокувати виклик, негайно повернути результат, якщо жоден відповідний процес ще не завершився (зупинився або продовжився);
* ***WUNTRACED*** - при його виборі встановлюється параметр ***WIFSTOPPED***, навіть якщо процес, що викликає, не відстежує свій дочірній; ця властивість допомагає реалізувати більш загальне управління завданнями, як це зроблено в оболонці;
* ***WCONTINUED*** - якщо встановлено, то біт ***WIFCONTINUED*** в повернутому параметрі статусу встановлюється навіть якщо процес, що викликає, не відстежує свій дочірній; як і у випадку з ***WUNTRACED***, параметр корисний для реалізації оболонки.

У разі успіху ***waitpid()*** повертає ***pid*** процесу, статус якого змінився. Якщо встановлений ***WNOHANG***, а вказаний дочірній процес (один або кілька) не змінив свій статус, ***waitpid()*** поверне ***0***. У разі помилки виклик повертає ***-1***, а ***errno*** приймає одне з таких трьох значень:

***ECHILD*** - процес або процеси, вказані за допомогою аргументу ***pid***, не існують або не є нащадками викликає;

***EINTR*** - параметр ***WNOHANG*** ні встановлено, а сигнал був отриманий під час очікування;

***EINVAL*** - аргумент ***options*** вказано некоректно.

Розглянемо приклад. Програма повинна отримати значення, що повертається дочірнім процесом з ***pid 1742***, причому зробити це негайно, якщо дочірній процес ще не завершився. Можна реалізувати це в такий спосіб:

***int status;***

***pid\_t pid;***

***pid = waitpid(1742, &status, WNOHANG);***

***if (pid == -1)***

***perror("waitpid");***

***else***

***{***

***printf("pid =%d\n", pid);***

***if (WIFEXITED(status))***

***printf("Звичайне завершення зі статусом виходу =%d\n",WEXITSTATUS(status));***

***if (WIFSIGNALED(status))***

***printf("Вбитий сигналом =%d%s\n", WTERMSIG(status), WCOREDUMP(status) ? "(Дамп ядра)": "");***

***}***

Останній приклад. Зверніть увагу на наступне використання ***wait()***:

***wait(&status);***

Це повністю аналогічно:

***waitpid(-1,&status,0);***

**Лекція №16**

**«Сигнали у Linux. Основи управління сигналами. Відправлення сигналів. Набори сигналів»**

Сигнали - це програмні переривання, що забезпечують асинхронну обробку подій. Ці події можуть приходити з-за меж системи; наприклад, користувач може згенерувати символ переривання, натиснувши ***<Ctrl>+<C>***. Інші джерела переривань - дії програми або ядра; наприклад, сигнал виникне, якщо процес виконає код, в якому відбувається поділ на нуль. В якості примітивної форми міжпроцесної комунікації (IPC) один процес також може послати сигнал іншому процесу.

Основна риса сигналів полягає в тому, що не тільки події відбуваються асинхронно - наприклад, користувач може натиснути ***<Ctrl>+<C>*** в будь-який момент роботи програми, - але й обробка сигналів в програмі виконується асинхронно. Функції обробки сигналів реєструються в ядрі, яке асинхронно викликає функції з іншої частини програми, коли програма отримує той чи інший сигнал.

Ще на зорі існування UNIX в цій операційній системі застосовувалися сигнали. Згодом вони розвивалися, причому найбільш значний прогрес спостерігається в підвищенні їх надійності (раніше сигнали могли губитися), а також функціональності, оскільки сучасні сигнали можуть нести корисне навантаження, яка визначається користувачем. Раніше різні системи UNIX вносили в роботу сигналів такі зміни, через які сигнали з різних систем могли ставати несумісними. На щастя, ця проблема була вирішена в рамках POSIX, який стандартизував обробку сигналів. Саме цей стандарт і використовується в Linux, про нього ми тут і поговоримо.

Самі нетривіальні додатки взаємодіють з використанням сигналів. Навіть якщо ви спеціально проектуєте ваш додаток таким чином, що вся його комунікація може виконуватися без опори на сигнали, в деяких випадках без сигналів просто не обійтися. Так, вони необхідні при завершенні програми.

***Концепції, пов'язані з сигналами***

Сигнали мають дуже суворий життєвий цикл. Спочатку сигнал породжується (іноді також кажуть, що він відсилається або генерується). Потім ядро ​​зберігає сигнал до тих пір, поки не зможе його доставити. Нарешті, як тільки з'являється така можливість, ядро ​​обробляє сигнал потрібним чином. Залежно від вимог процесу ядро ​​може виконати одну з трьох дій.

**1. Ігнорувати сигнал.** Ніяких дій не робиться. Є два сигнали, які не можуть бути проігноровані: ***SIGKILL*** і ***SIGSTOP***. Справа в тому, що системний адміністратор повинен мати можливість зупиняти або завершувати (вбивати) процеси. Якби процес був здатний проігнорувати ***SIGKILL*** (стати «безсмертним») або ***SIGSTOP*** (стати «нестримним»), то системний адміністратор позбувся б такої можливості.

**2. Перехопити сигнал і обробити його.** Ядро зупиняє виконання поточного коду в процесі і переключається на функцію, яка була зареєстрована раніше. Потім процес виконує цю функцію. Коли він повернеться після її виконання, процес знову перейде до виконання тієї роботи, яку перервав в момент отримання сигналу. Особливо часто доводиться відловлювати сигнали ***SIGINT*** і ***SIGTERM***. Процеси відловлюють ***SIGINT***, реагуючи на дію користувача, яка згенерувала символ переривання. Наприклад, термінал може перехопити цей сигнал і повернутися в основне вікно із запрошенням. Процеси відловлюють ***SIGTERM*** для виконання необхідної очищення, наприклад для від'єднання від мережі або для видалення тимчасових файлів - до завершення. Сигнали ***SIGKILL*** і ***SIGSTOP*** не можна перехопити.

**3. Виконання дії, що задається за замовчуванням.** Ця дія залежить від того, який саме сигнал був відправлений. Найчастіше стандартне дію - це завершення процесу. Наприклад, саме так обробляється сигнал ***SIGKILL***. Проте багато сигналів надаються для строго певної мети і цікавлять програміста лише в конкретній ситуації. Тому за замовчуванням подібні сигнали ігноруються, так як більшість програм в них «не зацікавлені». Нижче ми розглянемо різні сигнали і стандартні дії, що виконуються за замовчуванням при їх отриманні.

У колишні часи, коли сигнал був доставлений, обробити його функція не мала ніякої інформації про те, що ж сталося, - вона знала тільки про сам факт виникнення певного сигналу. В даний час ядро ​​може надавати зацікавленому програмісту широкий контекст того, що відбувається. Як ми незабаром побачимо, сигнали можуть передавати навіть дані, визначені користувачем.

***Ідентифікатори сигналів***

У кожного сигналу є символьне ім'я, що починається з префікса ***SIG***. Наприклад, ***SIGINT*** відсилається, якщо користувач натиснув ***<Ctrl>+<C>***. Сигнал ***SIGABRT*** генерується, коли процес викликає функцію ***abort()***. Нарешті, сигнал ***SIGKILL*** посилається при примусовому завершення процесу.

Всі ці сигнали описуються в заголовку, що включається з ***<signal.h>***. Сигнали - це просто препроцесорні визначення, що представляють позитивні цілі числа. Таким чином, кожен сигнал асоційований з цілочисельним ідентифікатором. У сигналів відображення імені на ціле число залежить від реалізації і відрізняється в різних системах UNIX. Правда, близько десятка найбільш поширених сигналів всюди відображаються однаково (наприклад, ***SIGKILL*** - це сумно відомий сигнал ***9***). У програмі кожен сигнал завжди повинен мати ім'я, зручне для читання, і ніколи - цілочисельне значення.

Номери сигналів починаються з 1 (зазвичай це ***SIGHUP***) і далі зростають. Всього існує близько 30 сигналів, але більшість програм регулярно використовують лише кілька з них. Сигналу із значенням 0 не існує; це спеціальне значення, відоме як нульовий сигнал. З нульовим сигналом не пов'язане анічогісінько цікавого - він не заслуговує власного імені. Але деякі системні виклики (наприклад, ***kill()***) використовують значення 0 в якості спеціального випадку.

***Сигнали, які підтримуються в Linux***

У табл. 16.1 перераховані сигнали, які підтримуються в Linux.

Табл. 16.1. Сигнали

|  |  |  |
| --- | --- | --- |
| **Сигнал** | **Опис** | **Дія за замовчуванням** |
| ***SIGABRT*** | Відправляється функцією ***abort()*** | Завершення з дампом ядра |
| ***SIGALRM*** | Відправляється функцією ***alarm()*** | Завершення |
| ***SIGBUS*** | Збій або помилка вирівнювання | Завершення з дампом ядра |
| ***SIGCHLD*** | Дочірній процес був завершений | Ігнорується |
| ***SIGCONT*** | Процес продовжив роботу і був зупинений | Ігнорується |
| ***SIGFPE*** | Арифметичне виключення | Завершення з дампом ядра |
| ***SIGHUP*** | Термінал, що управляє процесом, був закритий (як правило, при виході користувача з системи) | Завершення |
| ***SIGILL*** | Процес спробував виконати неприпустиму інструкцію | Завершення з дампом ядра |
| ***SIGINT*** | Користувач згенерував символ переривання (***<Ctrl>+<C>***) | Завершення |
| ***SIGIO*** | Подія асинхронного введення/виведення | Завершення |
| ***SIGKILL*** | Завершення процесу, що не можливо спіймати | Завершення |
| ***SIGPIPE*** | Процес записав інформацію в конвеєр, але зчитувачів не було | Завершення |
| ***SIGPROF*** | Минув час, заданий на профілюючому таймері | Завершення |
| ***SIGPWR*** | Збій в енергопостачанні | Завершення |
| ***SIGQUIT*** | Від користувача отриманий символ виходу (***<Ctrl>+<\>***) | Завершення з дампом ядра |
| ***SIGSEGV*** | Порушення доступу до пам'яті | Завершення з дампом ядра |
| ***SIGSTKFLT*** | Помилка в стеці співпроцесора | Завершення |
| ***SIGSTOP*** | Призупиняє виконання процесу | Зупинка |
| ***SIGSYS*** | Процес спробував виконати неприпустимий системний виклик | Завершення з дампом ядра |
| ***SIGTERM*** | Відловлювати завершення процесу | Завершення |
| ***SIGTRAP*** | Досягнута точка зупинки | Завершення з дампом ядра |
| ***SIGTSTP*** | Користувач згенерував символ призупинення (***<Ctrl>+<Z>***) | Зупинка |
| ***SIGTTIN*** | Фоновий процес виконав зчитування з керуючого терміналу | Зупинка |
| ***SIGTTOU*** | Фоновий процес виконав запис в керуючий термінал | Зупинка |
| ***SIGURG*** | Операція введення/виведення терміново вимагає обробки | Ігнорується |
| ***SIGUSR1*** | Сигнал, який визначається процесом | Завершення |
| ***SIGUSR2*** | Сигнал, який визначається процесом | Завершення |
| ***SIGVTALRM*** | Генерується функцією ***setitimer()*** при виклику з прапором ***ITIMER\_VIRTUAL*** | Завершення |
| ***SIGWINCH*** | Змінився розмір вікна керуючого терміналу | Ігнорується |
| ***SIGXCPU*** | Перевищено ліміт ресурсів, виділених процесу | Завершення з дампом ядра |
| ***SIGXFSZ*** | Перевищено ліміт ресурсів, виділених файлу | Завершення з дампом ядра |

***Основи управління сигналами***

Отже, ми розібралися зі значеннями всіх сигналів. Тепер розглянемо, як управляти ними всередині вашої програми. Найпростіший і найстаріший інтерфейс для роботи з сигналами - це функція ***signal()***. Вона визначається в стандарті **ISO C89**, який регламентує лише найменший «спільний знаменник» підтримки сигналів. Тому даний системний виклик дуже простий. Linux надає і інші інтерфейси, що забезпечують набагато більш повний контроль при роботі з сигналами. Оскільки функція ***signal()*** - найпростіша з собі подібних, а також описана в **ISO C** і тому досить поширена, ми для початку розглянемо саме її:

***#include <signal.h>***

***typedef void (\* sighandler\_t) (int);***

***sighandler\_t signal (int signo, sighandler\_t handler);***

Успішний виклик ***signal()*** скасовує актуальну дію, розпочату у відповідь на отримання сигналу ***signo***. Замість цього сигнал обробляється за допомогою механізму, зазначеного в ***handler***. ***signo*** - це назва одного з сигналів, розглянутих в попередньому розділі, наприклад ***SIGINT*** або ***SIGUSR1***. Як ви пам'ятаєте, процес не зможе перехопити сигнали ***SIGKILL*** і ***SIGSTOP***, тому встановлювати один з них в якості значення для обробника не має сенсу.

Функція ***handler*** повинна повертати ***void***. І це правильно, оскільки для цієї функції (на відміну від звичайних) в програмі не передбачено стандартне місце для повернення. Ця функція приймає один аргумент - ціле число, яке є ідентифікатором оброблюваного сигналу (наприклад, ***SIGUSR2***). Він дозволяє одній функції обробляти відразу багато сигналів. Прототип має наступну форму:

***void my\_handler (int signo);***

Для визначення цього прототипу Linux використовує оголошення ***sighandler\_t***. В інших системах UNIX безпосередньо задіяні вказівники функцій. В окремих системах ви знайдете власні типи, які можуть називатися не ***sighandler***\_t, а якось інакше. Якщо необхідно забезпечити переносимість програми, то в ній не слід безпосередньо посилатися на типи.

Коли ця функція надсилає сигнал до процесу, що зареєстрував обробник сигналів, ядро ​​призупиняє виконання основного потоку інструкцій програми і викликає обробник сигналу. Оброблювач отримує значення сигналу, яке представляє собою ***signo***, спочатку передане функції ***signal()***.

Ви також можете застосовувати функцію ***signal()***, щоб наказати ядру проігнорувати вказаний сигнал для актуального процесу або щоб скинути сигнал і застосувати поведінку, задане за замовчуванням. Це робиться за допомогою спеціальних значень, що присвоюються параметру ***handler***:

* ***SIG\_DFL*** - встановлює поведінку сигналу, зазначеного як ***signo***, як діючий за умовчанням. Наприклад, в разі ***SIGPIPE*** процес буде завершений;
* ***SIG\_IGN*** - ігнорує сигнал, зазначений як ***signo***.

Функція ***signal()*** повертає попередню поведінку сигналу. Це може бути застосування вказівника на обробник сигналу, значення ***SIG\_DFL*** або ***SIG\_IGN***. При помилці функція повертає ***SIG\_ERR***. Вона не встановлює значення ***errno***.

***Очікування будь-якого сигналу***

Стандарт **POSIX** визначає системний виклик ***pause()***, корисний при налагодженні і при написанні демонстраційних фрагментів коду. Цей виклик переводить процес в сплячий режим до тих пір, поки процес не отримає сигнал, який або буде оброблений, або завершить сам процес:

***#include <unistd.h>***

***int pause (void);***

Функція ***pause()*** повертається лише в тому випадку, якщо сигнал отриманий (і, відповідно, цей сигнал обробляється). У такій ситуації ***pause()*** повертає ***-1*** і присвоює ***errno*** значення ***EINTR***. Якщо ядро ​​породжує ігнорований даним процесом сигнал, то процес не прокидається.

У ядрі Linux ***pause()*** є одним з найпростіших системних викликів. Він виконує всього дві дії. По-перше, переводить процес в безперервний сплячий режим. По-друге, викликає ***schedule()***, щоб активізувати планувальник процесів Linux і знайти інший процес для запуску. Оскільки цей процес фактично нічого не очікує, ядро ​​не стане його «будити», поки він не отримає сигнал. Всі ці перипетії виражаються за все в двох рядках коду на **C**.

***Приклади***

Розглянемо пару простих прикладів. У першому з них ми реєструємо обробник сигналу ***SIGINT***, який просто виводить повідомлення, а потім завершує програму (як зазвичай при використанні ***SIGINT***):

***#include <stdlib.h>***

***#include <stdio.h>***

***#include <unistd.h>***

***#include <signal.h>***

***/\* обробник для SIGINT \*/***

***static void sigint\_handler (int signo)***

***{***

***printf ("Захоплений сигнал SIGINT!\n");***

***exit (EXIT\_SUCCESS);***

***}***

***int main (void)***

***{***

***/\* Реєструємо sigint\_handler як наш обробник для сигналу SIGINT \*/***

***if (signal (SIGINT, sigint\_handler) == SIG\_ERR)***

***{***

***fprintf (stderr, "Неможливо обробити SIGINT!\n");***

***exit (EXIT\_FAILURE);***

***}***

***for (;;)***

***pause ();***

***return 0;***

***}***

У наступному прикладі ми зареєструємо точно такий же обробник для ***SIGTERM*** і ***SIGINT***. Крім того, ми скинемо поведінку сигналу ***SIGPROF***, щоб він діяв, як встановлено за замовчуванням (процес буде завершуватися), і проіґноруємо ***SIGHUP*** (який в іншому випадку завершив би процес):

***#include <stdlib.h>***

***#include <stdio.h>***

***#include <unistd.h>***

***#include <signal.h>***

***/\* обробник для SIGINT и SIGTERM \*/***

***static void signal\_handler (int signo)***

***{***

***if (signo == SIGINT)***

***printf ("Захоплений сигнал SIGINT!\n");***

***else if (signo == SIGTERM)***

***printf ("Захоплений сигнал SIGTERM!\n");***

***else***

***{***

***/\* цього ні в якому разі не повинно статися \*/***

***fprintf (stderr, "Неочікуваний сигнал!\n");***

***exit (EXIT\_FAILURE);***

***}***

***exit (EXIT\_SUCCESS);***

***}***

***int main (void)***

***{***

***/\* Реєструємо signal\_handler як наш обробник сигналу для SIGINT \*/***

***if (signal (SIGINT, signal\_handler) == SIG\_ERR)***

***{***

***fprintf (stderr, "Неможливо обробити SIGINT!\n");***

***exit (EXIT\_FAILURE);***

***}***

***/\* Реєструємо signal\_handler як наш обробник сигналу для SIGTERM \*/***

***if (signal (SIGTERM, signal\_handler) == SIG\_ERR)***

***{***

***fprintf (stderr, " Неможливо обробити SIGTERM!\n");***

***exit (EXIT\_FAILURE);***

***}***

***/\* Скидаємо поведінку SIGPROF, щоб він діяв як за замовчуванням \*/***

***if (signal (SIGPROF, SIG\_DFL) == SIG\_ERR)***

***{***

***fprintf (stderr, "Неможливо скинути SIGPROF!\n");***

***exit (EXIT\_FAILURE);***

***}***

***/\* Ігноруємо SIGHUP \*/***

***if (signal (SIGHUP, SIG\_IGN) == SIG\_ERR)***

***{***

***fprintf (stderr, " Неможливо скинути SIGHUP!\n");***

***exit (EXIT\_FAILURE);***

***}***

***for (;;)***

***pause ();***

***return 0;***

***}***

***Відправлення сигналу***

Системний виклик ***kill()***, що лежить в основі поширеною утиліти ***kill***, відправляє сигнал від одного процесу до іншого:

***#include <sys/types.h>***

***#include <signal.h>***

***int kill(pid\_t pid, int signo);***

У звичайній ситуації (коли ***pid*** більше нуля) ***kill()*** посилає сигнал ***signo*** процесу з ідентифікатором ***pid***.

Якщо параметр ***pid*** дорівнює нулю, ***signo*** відправляється всім процесам, що належать до тієї ж групи, що і викликає процес.

Якщо ***pid*** дорівнює ***-1***, то сигнал ***signo*** відсилається всім тим процесам, яким викликає процес може відправити сигнал (має на це право доступу), за винятком ***init*** і себе самого. У наступному підрозділі ми обговоримо права доступу, керуючі доставкою сигналів.

Якщо ***pid*** менше ***-1***, то сигнал відсилається групі процесів ***-pid***.

У разі успіху ***kill()*** повертає ***0***. Виклик вважається успішним, як тільки вдасться послати один сигнал. При помилку (жоден сигнал послати не вдалося) виклик повертає ***-1*** і присвоює ***errno*** одне з наступних значень:

* ***EINVAL*** - сигнал, позначений ***signo***, є неприпустимим;
* ***EPERM*** - викликає, не володіє достатніми правами доступу, щоб послати сигнал якого-небудь запрошенням процесу;
* ***ESRCH*** - процес або група процесів, певні pid, не існує або (в разі процесу) це процес-зомбі.

Далі показаний приклад відправки сигналу ***SIGHUP*** процесу з ***ID 1722***:

***int ret;***

***ret = kill (1722, SIGHUP);***

***if (ret)***

***perror ("kill");***

Фактично цей фрагмент коду аналогічний наступному, що викликає утиліту ***kill***:

***$ kill -HUP +1722***

Щоб перевірити, чи є у нас право відправити сигнал процесу ***1722***, які не посилаючи при цьому сам сигнал, можна вчинити так:

***int ret;***

***ret = kill (1722, 0);***

***if (ret)***

***; /\* Право доступу відсутнє \*/***

***else***

***; /\* Право доступу є \*/***

***Відправлення сигналу самому собі***

За допомогою функції ***raise()*** процес легко може відправити сигнал сам собі:

***#include <signal.h>***

***int raise (int signo);***

Цей виклик:

***raise(signo);***

еквівалентний наступному:

***kill(getpid(), signo);***

У разі успіху цей виклик повертає 0, а при помилці - нульове значення. Він не встановлює змінну ***errno***.

***Відправлення сигналу цілій групі процесів***

Ще одна допоміжна функція дозволяє із зручністю відправляти сигнал всім процесам, що належать до вибраної групи, в тих випадках, коли заперечення ID групи процесів і використання ***kill()*** видається занадто витратною операцією:

***#include <signal.h>***

***int killpg (int pgrp, int signo);***

Цей виклик

***killpg (pgrp, signo);***

еквівалентний наступному:

***kill (-pgrp, signo);***

Виклик залишається вірним, навіть якщо ***pgrp*** дорівнює нулю. В такому випадку ***killpg()*** посилає сигнал ***signo*** кожному процесу з тієї групи, до якої відноситься процес, що викликає.

При успіху ***killpg()*** повертає ***0***. При помилці він повертає ***-1*** і присвоює ***errno*** одне з наступних значень:

* ***EINVAL*** - сигнал, зазначений як ***signo***, є неприпустимим;
* ***EPERM*** - викликає, не володіє достатніми правами доступу, щоб відправити сигнал всім необхідним процесам;
* ***ESRCH*** - група процесів, позначена ***pgrp***, не існує.

***Набори сигналів***

Окремим функціям потрібно маніпулювати наборами сигналів. Наприклад, процес може блокувати набір сигналів або набір сигналів може очікувати обробки. Для цього застосовуються операції з наборами сигналів:

***#include <signal.h>***

***int sigemptyset (sigset\_t \* set);***

***int sigfillset (sigset\_t \* set);***

***int sigaddset (sigset\_t \* set, int signo);***

***int sigdelset (sigset\_t \* set, int signo);***

***int sigismember (const sigset\_t \* set, int signo);***

Функція ***sigemptyset()*** ініціалізує набір сигналів, зазначений в ***set***, позначаючи його як порожня множина (з цього набору виключаються всі сигнали). Функція ***sigfillset()*** ініціалізує набір сигналів, зазначений в ***set***, позначаючи його як повний (всі сигнали включаються в набір). Обидві функції повертають ***0***. Ці дві функції потрібно застосувати до набору сигналів, перш ніж приступати до подальшого його використання.

Функція ***sigaddset()*** додає сигнал ***signo*** в набір, зазначений в ***set***, а ***sigdelset()*** видаляє ***signo*** з того набору сигналів, який вказаний в ***set***. Обидві функції повертають ***0*** в разі успіху або ***-1*** при помилці. У другому випадку змінної ***errno*** присвоюється код помилки ***EINVAL***, і це вказує, що ***signo*** є неприпустимим ідентифікатором сигналу.

Функція ***sigismember()*** повертає ***1***, якщо ***signo*** - це набір сигналів, зазначений в ***set***, ***0*** - якщо це інший набір, і ***-1*** - при помилці. В останньому випадку змінної ***errno*** присвоюється код ***EINVAL***, вказуючи, що значення ***signo*** є неприпустимим.

***Розширене управління сигналами***

Функція ***signal()***, з якою ми познайомилися на початку цієї глави, дуже проста. Оскільки вона входить до складу базової бібліотеки **C** і, отже, повинна відображати мінімальний набір припущень про можливості тієї операційної системи, в якій вона використовується, вона служить лише найменшим спільним знаменником при управлінні сигналами. В якості альтернативи POSIX також стандартизує системний виклик ***sigaction()***, що надає більш широкі можливості при управлінні сигналами. Зокрема, цю функцію можна застосовувати для того, щоб блокувати прийом зазначених сигналів в період дії вашого обробника, а також для отримання різноманітної інформації про систему і про стан процесу в момент породження сигналу:

***#include <signal.h>***

***int sigaction (int signo, const struct sigaction \* act, struct sigaction \* oldact);***

Виклик функції ***sigaction()*** змінює поведінку сигналу, позначеного аргументом ***signo***. Цей аргумент може приймати будь-які значення, крім тих, що асоціюються з ***SIGKILL*** і ***SIGSTOP***. Якщо ***act*** не дорівнює ***NULL***, то цей системний виклик змінює актуальну поведінку сигналу так, як це зазначено в ***act***. Якщо ***oldact*** не дорівнює ***NULL***, то виклик зберігає там попереднє (а в разі ***NULL*** - актуальне) поведінка зазначеного сигналу.

Структура ***sigaction*** забезпечує деталізований контроль над сигналами. Заголовок ***<sys/signal.h>***, що включається з ***<signal.h>***, визначає цю структуру в такий спосіб:

***struct sigaction***

***{***

***void (\* sa\_handler) (int); / \* Обробник сигналу або дії \* /***

***void (\* sa\_sigaction) (int, siginfo\_t \*, void \*);***

***sigset\_t sa\_mask; / \* Сигнали, які слід блокувати \* /***

***int sa\_flags; / \* Прапори \* /***

***void (\* sa\_restorer) (void); / \* Застаріле поле, не відповідає POSIX \* /***

***};***

В поле ***sa\_handler*** вказується дію, яке повинно бути здійснено після отримання сигналу. Як і у випадку з функцією ***signal()***, це поле може бути забезпечено прапором ***SIG\_DFL***, що позначає дію, задане за замовчуванням, прапором ***SIG\_IGN***, при якому ядро ​​ігнорує сигнал для процесу, або покажчиком на функцію, обробну сигнал. Ця функція має такий же прототип, як і обробник сигналів, що встановлюється функцією ***signal()***:

***void my\_handler (int signo);***

Якщо прапор ***SA\_SIGINFO*** встановлений в поле ***sa\_flags***, то саме поле ***sa\_sigaction***, а не ***sa\_handler*** визначає ту функцію, яка буде обробляти сигнал. Прототип цієї функції трохи відрізняється від попереднього:

***void my\_handler (int signo, siginfo\_t \* si, void \* ucontext);***

Ця функція отримує номер сигналу в якості свого першого параметра, структуру ***siginfo\_t*** - в якості другого і структуру ***ucontext\_t*** (наведену до вказівника ***void***) - в якості третьої параметра. Вона не має значення, що повертається. Структура ***siginfo\_t*** надає оброблювачу сигналів вичерпну інформацію, з якої ми познайомимося трохи нижче.

Зверніть увагу: в деяких машинних архітектурах (і, можливо, в інших системах UNIX) поля ***sa\_handler*** і ***sa\_sigaction*** знаходяться в об'єднанні, і ви не повинні привласнювати значення обох полях.

В поле ***sa\_mask*** надається набір сигналів, які система повинна блокувати протягом виконання обробника сигналу. Таким чином, програміст може самостійно накладати потрібний захист від повторного входу відразу для багатьох обробників сигналів. Сигнал, що обробляється в поточний момент, також блокується, якщо в поле ***sa\_flags*** не встановлено прапор ***SA\_NODEFER***. Ви не можете блокувати ***SIGKILL*** або ***SIGSTOP***. Виклик буде безшумно ігнорувати будь-який з них в ***sa\_mask***.

Поле ***sa\_flags*** - це бітова маска, що включає нуль, один або кілька прапорів, які змінюють обробку сигналу, зазначеного як ***signo***.

***структура siginfo\_t***

Структура ***siginfo\_t*** також визначається в ***<sys/signal.h>*** наступним чином:

***typedef struct siginfo\_t***

***{***

***int si\_signo; /\* Номер сигналу \*/***

***int si\_errno; /\* Значення errno \*/***

***int si\_code; /\* Код сигналу \*/***

***pid\_t si\_pid; /\* Ідентифікатор pid відсилає процесу \*/***

***uid\_t si\_uid; /\* Реальний ідентифікатор uid відсилає процесу \*/***

***int si\_status; /\* Значення виходу або сигнал \*/***

***clock\_t si\_utime; /\* Витрачений для користувача час \*/***

***clock\_t si\_stime; /\* Витрачений системне час \*/***

***sigval\_t si\_value; /\* Значення корисного навантаження сигналу \*/***

***int si\_int; /\* Сигнал POSIX.1b \*/***

***void \* si\_ptr; /\* Сигнал POSIX.1b \*/***

***void \* si\_addr; /\* Місце розташування в пам'яті, що спровокувало помилку \*/***

***int si\_band; /\* Подія смуги \*/***

***int si\_fd; /\* Дескриптор файлу \*/***

***};***

Ця структура рясніє інформацією, переданої оброблювачу сигналів (якщо ви використовуєте ***sa\_sigaction*** замість ***sa\_sighandler***). В сучасних обчислювальних системах сигнальна модель UNIX сприймається багатьма як дуже незручний спосіб виконання міжпроцесної комунікації. Ймовірно, такі скептики просто звикли застосовувати ***signal()*** в тих випадках, коли слід було б вдатися до ***sigaction()*** з прапором ***SA\_SIGINFO***. Структура ***siginfo\_t*** дозволяє вичавити з сигналів набагато більше функціональних можливостей.

У цій структурі чимало цікавих даних. Зокрема, тут ми знаходимо інформацію про процес, який відправив сигнал, і про причини цього сигналу. Нижче наводиться докладний опис кожного з полів цієї структури.

* ***si\_signo*** - номер сигналу, що нас цікавить. Перший аргумент вашого обробника сигналів також надає цю інформацію (що дозволяє уникнути розіменування вказівника);
* ***si\_errno*** - якщо це поле нульове, то воно містить код помилки, асоційований з цим сигналом. Поле дійсно для всіх сигналів;
* ***si\_code*** - описує, чому і звідки процес отримав цей сигнал (наприклад, від ***kill()***). У наступному розділі ми розглянемо всі можливі значення цього поля. Поле дійсно для всіх сигналів;
* ***si\_pid*** - для ***SIGCHLD*** надає pid-ідентифікатор того процесу, який був завершений;
* ***si\_status*** - для ***SIGCHLD*** надає статус виходу того процесу, який був завершений;
* ***si\_utime*** - для ***SIGCHLD*** надає витрачений для користувача час того процесу, який був завершений;
* ***si\_stime*** - для ***SIGCHLD*** надає витрачений системне час того процесу, який був завершений;
* ***si\_va***lue - об'єднання ***si\_int*** і ***si\_ptr***;
* ***si\_int*** - для сигналів, які були надіслані за допомогою функції ***sigqueue()***, надане навантаження типізується як ціле число.
* ***si\_ptr*** - для сигналів, які були надіслані за допомогою функції ***sigqueue()***, надане навантаження типізується як вказівник ***void***;
* ***si\_addr*** - для сигналів ***SIGBUS***, ***SIGFPE***, ***SIGILL***, ***SIGSEGV*** і ***SIGTRAP*** цей вказівник ***void*** містить номер помилки, що викликала проблему. Наприклад, в разі ***SIGSEGV*** в цьому полі міститься та адресу, в якому відбулося порушення доступу до пам'яті (отже, тут нерідко зустрічається ***NULL***!);
* ***si\_band*** - для сигналу ***SIGPOLL*** тут міститься пріоритетна інформація для того файлового дескриптора, яка вказана в ***si\_fd***;
* ***si\_fd*** - для сигналу ***SIGPOLL*** тут міститься дескриптор того файлу, чия операція завершилася.

Поля ***si\_value***, ***si\_int*** і ***si\_ptr*** особливо складні в роботі, так як процес може використовувати їх для передачі довільних даних іншому процесу. Отже, ви можете задіяти їх, щоб передавати як просте ціле число, так і покажчик на структуру даних (до речі, в даному випадку покажчик вам практично не допоможе, якщо процеси не поділяють загального адресного простору). Ці поля докладно розглядаються в розд. «Відправлення сигналу з корисним навантаженням» далі.

POSIX гарантує, що лише три поля будуть дійсні для всіх сигналів. Доступ до інших полях повинен відбуватися лише при обробці застосовується з ними сигналу. Наприклад, ви повинні звертатися до поля ***si\_fd*** тільки при обробці сигналу ***SIGPOLL***.

**Лекція №17**

**«Багатопотоковість у Linux. Основи багатопотокового програмування у Linux на мові C/C++»**

**Потоковість** - це створення і управління безліччю виконуваних елементів всередині одного процесу. Потік є єдиним джерелом програмних помилок за рахунок впровадження конкурентності даних і клінчів (взаємних блокувань). Ми розглянемо основи потоковості АРI в Linux, щоб розібратися, як потоковість вписується в системний інструментарій програміста; коли потрібно використовувати потоки і, що важливіше, коли не потрібно; які шаблони проектування можуть допомогти спланувати і побудувати потокові додатки; що таке конкуренція даних і як її можна запобігти.

***Бінарні модулі, процеси і потоки***

**Бінарні модулі** - це програми, що знаходяться в пасивному стані в сховище і скомпільовані в форматі, прийнятному для даної операційної системи і машинної архітектури, готові до запуску, але поки недіючі. **Процеси** - це абстракції операційної системи, що представляють собою дані модулі в дії: завантажений бінарний код, віртуальна пам'ять, ресурси ядра, такі як відкриті файли, пов'язаний користувач і т. д. Потік - це елемент виконання всередині процесу: віртуальний процесор, стек або статус програми. Іншими словами, процеси - це виконуються бінарні модулі, і потоки є найменшими виконуваними елементами, передбаченими планувальником операційної системи.

Процес містить один або кілька потоків. Якщо процес містить тільки один потік, то в процесі знаходиться лише один виконуваний елемент і тільки одне завдання виконується в одиницю часу. Такі процеси можна назвати **однопотоковими**. Це класичні процеси UNIX. Якщо процес містить більше одного потоку, значить одночасно виконується кілька дій. такі процеси називаються **багатопотоковими**.

Сучасні операційні системи представляють в просторі користувача дві основні віртуальні абстракції - віртуальну пам'ять і віртуальний процесор. Спільно вони створюють для кожного процесу, що виконується, ілюзію, що він один користується всіма ресурсами машини. Віртуальна пам'ять надає кожному процесу унікальний вид пам'яті, який нібито відповідає оперативної пам'яті або сховища на диску (це досягається за допомогою розбивки на сторінки). Оперативна пам'ять системи може в дійсності містити дані сотні різних процесів, запущених одночасно, але кожному з них здається, що вся пам'ять належить тільки йому. Віртуальний процесор дозволяє кожному процесу працювати так, наче він один в системі, а операційна система приховує, що кілька процесів працюють одночасно на (можливо) декількох процесорах.

Віртуальна пам'ять пов'язується з процесом, а не з потоком. Таким чином, кожен процес має унікальний баченням пам'яті, але все потоки в даному процесі поділяють цю пам'ять. І навпаки, віртуальний процесор пов'язаний саме з потоками, а не з процесами. Кожен потік - індивідуально заплановане дію, що дозволяє кожному процесу виконувати більш одного дії в кожен момент часу. Багато програмістів поєднують дві ілюзії віртуальної пам'яті і віртуального процесора, але потоки вимагають розділяти їх. У потоків, як і у процесів, повинна бути ілюзія, що процесор (або декілька) цілком належить їм. Потоки, на відміну від процесів, не потребують будь-яких ілюзій, що їм належить вся пам'ять, - все потоки всередині процесу спільно використовують наявне в їх розпорядженні адресний простір.

***Багатопотоковість***

У чому суть потоків? Процеси зазвичай потрібні тому, що вони є абстракцією запущеної програми. Однак навіщо розділяти елементи виконання і вводити поняття потоків? Ось шість основних переваг багатопотоковості.

* **Програмна абстракція.** Поділ праці та призначення кожному дивізіону виконуваного елемента (потоку) - природний підхід при вирішенні багатьох завдань. В шаблони проектування, які використовують цей підхід, входять «один потік на одне з'єднання» і «пул потоків». Програмісти зазвичай вважають ці шаблони корисними і інтуїтивно зрозумілими. Деякі, однак, розглядають ці потоки як антишаблони. Таким чином, в теорії немає завдань, які можна розв'язати за допомогою потоків і не можна вирішити за допомогою кінцевих автоматів.
* **Паралелізм.** У машинах з декількома процесорами потоки забезпечують ефективний спосіб досягнення справжнього паралелізму. Кожен потік отримує власний віртуальний процесор, де його планування не залежить ні від чого, тому кілька потоків можуть виконуватися на декількох процесорах в один і той же час, покращуючи пропускну здатність системи.
* **Поліпшення реагування.** Навіть на однопроцесорній машині багатопотоковість може прискорити реагування процесів. При однопотоковому процесі тривалий час виконується операція може перешкодити додатком зреагувати на якусь дію користувача, якому може здатися, що додаток «зависло». При многопоточности такі операції можуть бути делеговані робочим потокам, дозволивши хоча б одному потоку залишатися здатним реагувати на дії користувача і виконувати операції введення-виведення.
* **Блокування введення-виведення.** Це пов'язано з попереднім пунктом. За відсутності потоків блокування введення-виведення зупиняє весь процес. Це може згубно впливати як на загальну пропускну здатність, так і на час затримки. У багатопотоковому процесі індивідуальні потоки можуть блокуватися, очікуючи введення-виведення, поки інші потоки продовжують роботу. Асинхронність і неблокування введення-виведення можуть бути альтернативними рішеннями цього завдання.
* **Перемикання контексту.** Витрати на переключення від одного потоку до іншого всередині того ж самого процесу значно менше, ніж переміщення контексту від одного процесу до іншого.
* Е**кономія пам'яті.** Потоки забезпечують ефективний спосіб поділу пам'яті, одночасно дозволяючи виконання декількох елементів. Таким чином вони є альтернативою декільком процесам.

За перерахованих вище причин потоковість досить поширена в різних операційних системах і їх додатках. У деяких системах, таких як Android, майже кожен процес діє за допомогою декількох потоків. 10-15 років тому твердження «потоки потрібні людям, які не можуть програмувати кінцеві автомати» було правдою, оскільки більшість переваг потоковості можна було реалізувати за допомогою інших засобів, таких як неблокування введення-виведення, і, так, кінцевих автоматів. Сьогодні кількість процесорів навіть на невеликих машинах - мобільні пристрої зараз також мають по кілька процесорів - і технології (багатоядерність і одночасна багатопотоковість) створюють необхідність в потоковості для максимального збільшення пропускної здатності в системному програмуванні. В наш час неможливо уявити собі високопродуктивний веб-сервіс, який працює без кількох потоків на декількох ядрах.

***Недоліки багатопотоковості***

Незважаючи на всі свої переваги, багатопотоковість все-таки пов'язана з певними вадами. Зокрема, одна з найстрашніших, найпідступніших помилок в історії програмування була пов'язана саме з потоковістю. Розробка, написання, розуміння і - найповчальніше - налагодження багатопотокових програм значно складніше, ніж однопотокових процесів.

Джерело небезпеки потоків безпосередньо полягає в сенсі їх існування: кілька віртуальних процесорів існують при тільки одному екземплярі віртуальної пам'яті. Іншими словами, при багатопотоковому процесі виконується одночасно кілька дій (паралелізм), які поділяють одну і ту ж область пам'яті. Таким чином, неминуче, що потоки в процесі ділять між собою ресурси - скажімо, їм потрібно вважати або записати одні й ті ж дані. Розуміння, як працює ваша програма, таким чином, переходить від простого викладу послідовності виконання інструкцій до концептуалізації декількох потоків, які виконуються незалежно один від одного, з непередбачуваним вибором часу виконання та інструкцій, необхідних для правильної роботи. Збій в синхронізації може перевести до перекриття виведення, некоректного виконання і припинення роботи програми. Розуміння та налагодження багатопотокових програм досить складна, тому вкрай важливо, щоб ваша потокова модель і стратегія синхронізації були частиною системного дизайну з самого початку.

***Альтернативи багатопотоковості***

Залежно від ваших цілей, пов'язаних з багатопотоковістю, можливі альтернативи. Наприклад, вигоди в затримці виконання і введення-виведення також досяжні за допомогою мультиплексного введення-виведення, введення-виведення, яке не блокується і асинхронного введення-виведення. Ці техніки дозволяють процесам здійснювати операції введення-виведення, що не блокують процес. Якщо ж вашою метою є істинний паралелізм, то N процесів можуть використовувати процесор так само, як N потоків, з урахуванням збільшення споживання ресурсів і витрат на переключення. І навпаки, якщо ваша мета - економія пам'яті, Linux забезпечує кошти для поділу пам'яті, що працюють в більш обмеженою манері, ніж потоки.

Сучасні системні програмісти, втім, не знаходять ці альтернативи адекватними. Асинхронне введення-виведення, наприклад, часто просто виводить з себе. І навіть якщо ви можете компенсувати витрати на кілька процесів за рахунок поділу пам'яті і інших ресурсів, витрати на перемикання нікуди не подінуться. Таким чином, потоки широко використовуються не тільки в системному програмуванні, а й взагалі в стеках: від ядра до додатків з графічним інтерфейсом. Через все більшого поширення багатоядерності використання потоків буде тільки частішати.

***Реалізація потоковості в Linux***

Ядро Linux забезпечує тільки базові примітиви для забезпечення поточности, наприклад системний виклик ***clone()***. Основна частина будь-якої потокової бібліотеки знаходиться в просторі користувача. Багато великих проектів по розробці ПЗ визначають власну потокову бібліотеку: наприклад, Android, Apache, GNOME і Mozilla забезпечують власні бібліотеки; крім того, мови програмування, такі як **С++ 11** і **Java**, забезпечують стандартну бібліотечну підтримку для потоків. Проте **POSIX** регламентує потокову бібліотеку з IEEE Std 1003.1c-1995 року, також відому як POSIX 1 995 або POSIX.1c. Розробники називають цей стандарт **POSIX-потоками**, або, для стислості, **Р-потоками**. Р-потоки залишаються лідируючим рішенням щодо потоковості для С і С++ в системах UNIX.

P-потоки як стандарт - це всього лише набір слів на сторінці. У Linux реалізація цього стандарту здійснюється за допомогою **glibc**, бібліотеки **С Linux**. Згодом **glibc** забезпечила дві різні реалізації Р-потоків: **LinuxThreads** і **NPTL**.

**LinuxThreads** - це оригінальна реалізація Р-потоків, що забезпечує потоковість 1:1. Вона вперше була включена в **glibc** версії 2.0, хоча і раніше була доступна як зовнішня бібліотека. **LinuxThreads** була розроблена для ядра, яке забезпечувало досить слабку підтримку потоковості: крім системного виклику ***clone()***, який створював новий потік, **LinuxThreads** реалізувала потоковість **POSIX**, використовуючи існуючі інтерфейси UNIX. Наприклад, **LinuxThreads** обробляє зв'язок «потік-на-потік» з використанням сигналів. Через відсутність підтримки Р-потоків з боку ядра в Linux реалізації **LinuxThreads** була потрібна наявність керуючого діями «менеджерського» потоку, який погано справлявся з великою кількістю потоків і не цілком відповідав стандарту **POSIX**.

**Native POSIX Thread Library (NPTL)** замінила **LinuxThreads** і залишається стандартною реалізацією Р-потоків в Linux. Вона була представлена ​​в Linux 2.6 і glibc 2.3. Як і **LinuxThreads**, **NPTL** забезпечує потоковість 1:1, що грунтується на системному виклику ***clone()*** і моделі ядра, де потоки розглядаються як будь-які інші процеси, за винятком здатності потоків розділяти між собою деякі ресурси. На відміну від **LinuxThreads**, **NPTL** отримує вигоду з додаткових інтерфейсів ядра, нових для версії 2.6, що включають системний виклик ***futex()*** для синхронізації потоків, системний виклик ***exit\_group()*** для припинення всіх потоків в процесі і підтримку ядра для локального поточного сховища (TLS). Таким чином, в NPTL вирішена проблема **LinuxThreads** з невідповідністю і значно покращено потокове масштабування, завдяки чому можливе створення тисяч потоків в одиничному процесі без будь-якого уповільнення.

***Створення потоків***

При першому запуску вашої програми і виконання функції ***main()*** програма є однопотоковому. Дійсно, за винятком забезпечення компілятором деяких параметрів безпеки потоку і наявності прив'язки до бібліотеки Р-потоків, ваш процес нічим не відрізняється від будь-якого іншого. З цього початкового потоку, який іноді називають потоком за замовчуванням або головним потоком, ви можете створити один або кілька додаткових, щоб запустити багатопотоковість.

Р-потоки забезпечують одну функцію для визначення і запуску нового потоку, яка називається ***pthread\_create ()***:

***#include <pthread.h>***

***int pthread\_create (pthread\_t \* thread,***

***const pthread\_attr\_t \* attr,***

***void \* (\* start\_routine) (void \*),***

***void \* arg);***

При успішному виконанні створюється новий потік і починає виконувати функцію, представлену в ***start\_routine*** з єдиним аргументом ***arg***. Функція збереже ідентифікатор потоку, використаний для представлення нового потоку, в ***pthread\_t***, зазначеного за допомогою ***thread***, якщо він не дорівнює ***NULL***.

Об'єкт ***pthread\_attr\_t***, який визначається через ***attr***, використовується для зміни атрибутів потоку, що привласнюються за замовчуванням, для новостворюваного потоку. Більшість викликів ***pthread\_create()*** відправляє в якості аргументу ***NULL***, привласнюючи атрибути за замовчуванням. Атрибути потоку дозволяють програмам змінювати безліч властивостей потоку, таких як розмір стека, параметри планування, початковий окремий статус. Повний обговорення атрибутів потоку виходить за рамки цієї глави, але man-сторінки Р-потоків - хороший ресурс.

Функція ***start\_routinе*** повинна мати запис наступного виду:

***void \* start\_thread (void \* arg);***

Таким чином потік починає існування з виконання функції, яка приймає вказівник ***void*** як єдиний аргумент, а потім повертає його як свою величину, що повертається. Аналогічно ***fork()*** новий потік успадковує більшість атрибутів, сумісність і статус від свого батька. Однак на відміну від ***fork()*** потоки поділяють ресурси зі своїм батьком замість отримання копії. Найважливішим ресурсом є, звичайно ж, адресний простір процесу, але потоки також ділять (замість отримання копії) обробники сигналів і відкриті файли.

У разі помилки ***pthread\_create()*** повертає ненульовий код помилки безпосередньо (без використання ***errno***) і вміст потоку ***thread*** не визначений. Помилки можуть включати в себе:

* ***EAGAIN*** - зухвалому процесу істотно бракує ресурсів для створення нового потоку; зазвичай це викликано тим, що процес досяг межі кількості потоків для кожного користувача або для всієї системи;
* ***EINVAL*** - об'єкт ***pthread\_attr\_t***, зазначений через ***attr***, включає неприпустимі атрибути;
* ***EPERM*** - викликає, не має повноважень для установки деяких атрибутів об'єкта ***pthread\_attr\_t***, зазначеного через ***attr***.

Приклад використання:

***pthread\_t tread;***

***int ret;***

***ret = pthread\_create (&thread, NULL, start\_routine, NULL);***

***if (!ret)***

***{***

***errno = ret;***

***perror("pthread\_create");***

***return -1;***

***}***

***Ідентифікатори потоків***

**Ідентифікатори потоків (TID)** для потоків є аналогами **ідентифікаторів процесів (PID)**. У той час як **PID** призначаються ядром Linux, **TID** призначаються лише бібліотекою Р-потоків. Цей тип представлений ***pthread\_t***, і **POSIX** не вимагає, щоб він був арифметичним. Як ми вже знаємо, **TID** нового потоку визначається за допомогою аргументу ***thread*** при успішному виклику ***pthread\_create()***. Потік може отримати свій **TID** при запуску за допомогою функції ***pthread\_self()***:

***#include <pthread.h>***

***pthread\_t pthread\_self(void);***

Використовувати функцію дуже просто, так як вона завжди працює успішно:

***const pthread\_t me = pthread\_self ();***

***Порівняння ідентифікаторів потоків***

Стандарт Р-потоків не вимагає, щоб ***pthread\_t*** був арифметичного типу, тому немає гарантії, що оператор рівності буде працювати. Отже, щоб порівняти ідентифікатори потоків, бібліотеці Р-потоків потрібен спеціальний інтерфейс:

***#include <pthread.h>***

***int pthread\_equal(pthread\_t t1, pthread\_t t2);***

Якщо наведені ідентифікатори потоків рівні, ***pthread\_equal()*** повертає ненульову величину. Якщо вони не рівні, повертається ***0***; помилка не може відбутися. Ось простий приклад:

***int ret;***

***ret = pthread\_equal(thing1, thing2);***

***if (ret! = 0)***

***printf("The TIDs are equal!\n");***

***else***

***printf("The TIDs are unequal!\n");***

***Завершення потоків***

Природною протилежністю створення потоків є їх завершення. Завершення потоків дуже схоже на завершення процесів, за винятком того, що, коли потік завершується, інші потоки в процесі продовжують виконуватися. У деяких поточних шаблонах, таких як потік на з'єднання, потоки часто створюються і знищуються.

Потоки можуть перериватися за певних обставин, які мають аналоги в завершенні процесів:

* якщо потік повертається з стартової процедури, він переривається; це аналог «виходу за межі» в ***main()***;
* якщо потік викликає функцію ***pthread\_exit()***, він завершується; це аналог виклику ***exit()***;
* якщо потік скасовується іншим потоком через функцію ***pthread\_cancel()***, він завершується; це аналог відправки сигналу ***SIGKILL*** через ***kill()***.

У цих трьох прикладах завершується тільки потік, на який спрямована дія. Всі потоки в процесі завершуються, зупиняючи таким чином сам процес, при наступних обставинах:

* процес повертається зі своєї функції ***main()***;
* процес завершується через ***exit()***;
* процес виконує новий двійковий образ через ***execve()***.

Сигнали можуть вбити процес або окремий потік в залежності від того, як вони спрямовані. Р-потоки роблять обробку сигналів трохи складніше, тому краще мінімізувати використання сигналів в бататопотокових програмах.

***Самозавершення***

Найпростіший шлях для потоку, щоб завершити самого себе, - це «вихід за межі» своєї початкової процедури. Однак часто вам буде потрібно завершити потік десь в глибині стека виклику функції, далеко від вашої стартової процедури. Для таких випадків в Р-потоках є виклик ***pthread\_exit()***, потоковий еквівалент ***exit()***:

***#include <pthread.h>***

***void pthread\_exit (void \* retval);***

За виконання потік, що викликає, завершується; ***retval*** забезпечується для кожного потоку, що очікує завершення, аналогічно ***exit()***. Помилка не може відбутися.

Використання:

***pthread\_exit (NULL);***

***Завершення інших потоків***

Р-потоки викликають завершення інших потоків через їх скасування. Це забезпечує функція ***pthread\_cancel()***:

***#include <pthread.h>***

***int pthread\_cancel (pthread\_t thread);***

Успішний виклик ***pthread\_cancel()*** надсилає запит на скасування потоку, представленому через ідентифікатор потоку ***thread***. Чи може потік бути скасований і коли, залежить від його стану відміни і типу скасування відповідно. У разі успіху ***pthread\_cancel()*** повертає ***0***. Зверніть увагу, що успіх в даному випадку означає лише успішну обробку запиту на скасування. Насправді ж завершення відбувається асинхронно. У разі помилки ***pthread\_cancel()*** повертає ***ESRCH***, що означає, що значення ***thread*** неприпустимо.

Умови, при яких потік може бути скасований, не такі прості. Стан відміни потоку може бути доступний або недоступний. За замовчуванням воно є доступним для нових потоків. З іншого боку, тип скасування вказує, коли відбувається скасування. Потоки можуть змінювати свій стан через ***pthread\_setcancelstate()***:

***#include <pthread.h>***

***int pthread\_setcancelstate(int state, int \* oldstate);***

У разі успіху стан відміни викликає потоку встановлюється на ***state***, а попередній стан зберігається в ***oldstate***. Значним ***state*** може бути ***PTHREAD\_CANCEL\_ENABLE*** або ***PTHREAD\_CANCEL\_DISABLE*** для дозволу або заборони скасування відповідно.

У разі помилки ***pthread\_setcancelstate()*** повертає ***EINVAL***, що означає неприпустиме значення ***state***.

Тип скасування потоку може бути асинхронним або відкладеним; за замовчуванням зазвичай встановлений останній. З асинхронним типом скасування потік може бути убитий в будь-якій точці після отримання команди на скасування. З відкладеним типом потік може бути убитий тільки в спеціальних точках скасування, які є функціями Р-потоків або бібліотеки **С** і являють собою безпечні моменти, в яких викликає потік може бути перерваний. Асинхронна скасування може бути корисна лише в певних ситуаціях, так як вона може залишити процес в невизначеному стані. Наприклад, що, якщо скасовувати потік був десь в середині критичної області? Щоб програма вела себе коректно, асинхронне скасування повинне використовуватися тільки потоками, для яких не передбачено спільне використання будь-яких ресурсів і ви можете викликати лише сигнально-безпечні функції. Потоки можуть змінити свій тип через ***pthread\_setcanceltype()***:

***#include <pthread.h>***

***int pthread\_setcanceltype (int type, int \* oldtype);***

У разі успіху статус скасування викликає потоку встановлюється в ***type***, а старий тип зберігається в ***oldtype***. Значним ***type*** може бути ***PTHREAD\_CANCEL\_ASYNCHRONOUS*** або ***PTHREAD\_CANCEL\_DEFERRED*** для установки асинхронної або відкладеної скасування відповідно.

У разі помилки ***pthread\_setcanceltype()*** повертає ***EINVAL***, що означає неприпустиме значення ***type***.

Розглянемо приклад, коли один потік повинен завершити інший. Спочатку потік, який повинен завершитися, дозволяє свою скасування і встановлює її тип як відкладений (ці значення встановлені за замовчуванням, так що в даному випадку це просто приклад):

***int unused;***

***int ret;***

***ret = pthread\_setcancelstate (PTHREAD\_CANCEL\_ENABLE, & unused);***

***if (ret)***

***{***

***errno = ret;***

***perror("pthread\_setcancelstate");***

***return -1;***

***}***

***ret = pthread\_setcanceltype(PTHREAD\_CANCEL\_DEFERRED, & unused);***

***if (ret)***

***{***

***errno = ret;***

***perror ("pthread\_setcanceltype");***

***return -1;***

***}***

Потім інший потік посилає команду на завершення:

***int ret;***

***/\* 'Thread' в даному випадку означає ідентифікатор потоку, що завершується \*/***

***ret = pthread\_cancel(thread);***

***if (ret)***

***{***

***errno = ret;***

***perror("pthread\_cancel");***

***return -1;***

***}***

***Приєднання і від'єднання потоків***

З огляду на, що потоки досить просто створюються і знищуються, повинен бути і спосіб їх синхронізувати замість завершення інших потоків - еквівалент ***wait()*** для потоковості. Дійсно, він існує. Це приєднання потоків.

***Приєднання потоків***

Приєднання дозволяє одному з потоків блокуватися в очікуванні завершення іншого:

***#include <pthread.h>***

***int pthread\_join(pthread\_t thread, void \*\* retval);***

Після успішного виконання викликає потік блокується до тих пір, поки потік, вказаний як ***thread***, що не завершиться (якщо ***thread*** вже завершено, ***pthread\_join()*** повертається негайно). Як тільки ***thread*** завершується, потік, що викликає, активізується і, якщо ***retval*** не дорівнює ***NULL***, отримує значення, що повертається завершеного процесу, передане ***pthread\_exit()*** або повернене від його стартової процедури. Після цього можна сказати, що потоки приєдналися один до одного. Приєднання завжди дозволяє потокам синхронізувати своє виконання по відношенню до періоду існування інших потоків. Всі потоки в Р-потоках є рівноправними; кожен потік може приєднуватися до будь-якого іншого. Один потік може приєднуватися до багатьох (фактично, як ми скоро побачимо, найчастіше один головний потік очікує інших потоків, які сам і створив), але тільки один потік може намагатися приєднатися до певного іншому, кілька потоків не повинні намагатися приєднатися до якого-небудь одному.

У разі помилки ***pthread\_join()*** повертає один з наступних ненульових кодів помилок:

* ***EDEADLK*** - сталось взаємне блокування - ***thread*** вже чекає приєднання до потоку, що викликає, або сам є потоком, що викликає;
* ***EINVAL*** - неможливо приєднати потік, визначений через ***thread***;
* ***ESRCH*** - значення ***thread*** неприпустимо.

Приклад використання:

***int ret;***

***/\* Приєднуємо до 'thread' і більше не дбаємо про величину, що повератється \*/***

***ret = pthread\_join (thread, NULL);***

***if (ret)***

***{***

***errno = ret;***

***perror("pthread\_join");***

***return -1;***

***}***

***Від'єднання потоків***

За замовчуванням потоки створюються здатними до приєднання. Однак вони можуть і від'єднуватися, але в цьому випадку вони стануть в подальшому не здатними бути приєднаними. Оскільки до приєднання потоки споживають будь-які системні ресурси, як роблять це і процеси, поки їхні предки викликають ***wait()***, потоки, які ви не плануєте приєднувати, повинні бути від'єднані.

***#include <pthread.h>***

***int pthread\_detach (pthread\_t thread);***

У разі успіху ***pthread\_detach()*** від'єднує потік, вказаний як ***thread***, і повертає ***0***. Результати не визначені, якщо ви викликаєте ***pthread\_detach()*** щодо потоку, який вже від'єднано. У разі помилки функція повертає значення ***ESRCH***, що означає, що значення ***thread*** неприпустимо.

Для кожного потоку в процесі необхідно викликати ***pthread\_join()*** або ***pthread\_detach()***, щоб системні ресурси могли вивільнитися після завершення потоку (звичайно, після того як завершується весь процес, все потокові ресурси вивільняються, але приєднання або від'єднання всіх процесів в явній формі залишається хорошою практикою).

***Приклад потоковості***

Наступний приклад повної програми з'єднає всі інтерфейси, розглянуті вище. Програма створює два потоки (всього їх буде три), починаючи обидва в одній і тій же стартовою процедурі ***start\_thread()***. Поведінка потоків у стартовій процедурі відрізняється аргументами. Потім обидва потоку приєднуються один до одного; якби цього не сталося, головний потік міг би завершитися до інших, перервавши весь процес.

***#include <stdlib.h>***

***#include <stdio.h>***

***#include <pthread.h>***

***void \* start\_thread (void \* message)***

***{***

***printf ("%s\n", (const char \*) message);***

***return message;***

***}***

***int main (void)***

***{***

***pthread\_t thing1, thing2;***

***const char \* message1 = "Thing 1";***

***const char \* message2 = "Thing 2";***

***/\* Створюються два потоки, кожен зі своїм повідомленням \*/***

***pthread\_create (&thing1, NULL, start\_thread, (void\*) message1);***

***pthread\_create (&thing2, NULL, start\_thread, (void\*) message2);***

***/\* Очікування завершення потоків. Якщо ми не приєднаємо їх тут, то ризикуємо знищити головний потік до того, як інші потоки завершаться \*/***

***pthread\_join(thing1, NULL);***

***pthread\_join(thing2, NULL);***

***return 0;***

***}***

Ось і вся програма. Якщо ви збережете її як ***example.c***, то можете скомпілювати її за допомогою такої команди:

***gcc -Wall -O2 -pthread example.c -o example***

Потім запустити наступним чином:

***./example***

На виході ви отримаєте:

***Thing 1***

***Thing 2***

Або, можливо:

***Thing 2***

***Thing 1***

Однак непотрібної інформації ніколи не буде. Чому? Тому що функція ***printf()*** безпечна для потоковості.