ЛЕКЦІЯ 19. ВВІД І ВИВІД ДАНИХ В UNIX-ПОДІБНИХ ОС.

Загальні положення

Будь-яка програма обробляє дані: одержуючи на вході одну інформацію, вони в результаті роботи видають іншу. Хоча вхідна й/або вихідна інформація може бути нульовою, тобто попросту відсутньою. Ті дані, які передаються програмі для обробки - це її *уведення*, те, що вона видає в результаті роботи - *вивід*. Організація *уведення* й *виводу* для кожної програми - це завдання операційної системи.

Кожна програма працює з даними певного типу: текстовими, графічними, звуковими т.п. Як, напевно, уже стало зрозуміло з попередніх лекцій, основний інтерфейс керування системою в Linux - це *термінал*, що призначений для передачі текстової інформації від користувача системі й назад (див. лекцію 2). Оскільки ввести з *термінала* вивести на *термінал* можна тільки текстову інформацію, то *уведення* й *вивід* програм, пов'язаних з терміналом[1)],теж повинен бути текстовим. Однак необхідність оперувати з текстовими даними не обмежує можливості керування системою, а, навпаки, розширює їх. Людина може **прочитати вивід** будь-якої програми й розібратися, що відбувається в системі, а різні програми виявляються сумісними між собою, оскільки використовують той самий вид подання даних - текстовий. Можливостям, які дає Linux при роботі з даними в текстовій формі, і присвячена дана лекція.

"Текстовість" даних - усього лише **домовленість** про їхній формат. Ніхто не заважає виводити на екран нетекстовий файл, однак користі в цьому буде мало. По-перше, раз вуж файл містить не текст, то не передбачається, що людина зможе що-небудь зрозуміти з його вмісту. По-друге, якщо в нетекстових даних, виведених на
термінал, **випадково** зустрінеться *керуюча послідовність*, *термінал* її виконає. Наприклад, якщо в скомпільованій програмі записане число 1528515121, воно представлено у вигляді чотирьох байтів: 27, 91, 49 і 74. Відповідний ним **текст** складається із чотирьох *символів* ASCII: "esc", "[", "1" і "J", і при *виводі* файлу на віртуальну консоль Linux у цьому місці виконається очищення екрана, тому що "^[[1J" - саме така *керуюча послідовність* для віртуальної консолі. Не всі *керуючі послідовності* настільки необразливі, тому використовувати нетекстові дані як тексти не рекомендується.

Перенапрямлення вводу і виводу

Для того щоб записати дані у **файл** або прочитати їх звідти, процесу необхідно спочатку **відкрити** цей файл (при відкритті на запис, можливо, прийде попередньо створити його) . При цьому процес одержує дескриптор (описатель) відкритого файлу - унікальне для цього процесу число, що він і буде використовувати у всіх операціях запису. Перший відкритий файл одержить дескриптор 0, другий - 1 і так далі. Закінчивши роботу з файлом, процес **закриває** його, при цьому дескриптор звільняється й може бути використаний повторно. Якщо процес завершується, не закривши файли, за нього це робить система. Строго говорячи, тільки в операції відкриття дескриптора вказується, який саме файл буде задіяний. У якості "файлу"

використовуються й звичайні файли, і файли-*дірки* (найчастіше - *термінали*), і *канали*, описані в розділі "*Конвеєр*". Подальші операції - читання, запис і закриття - працюють із дескриптором, як **з потоком даних,** а куди саме веде цей потік, неважливо.

Кожний процес Linux одержує при старті **три** "файли", відкритих для нього системою. Перший з них (дескриптор 0) відкритий на **читання**, це стандартне *уведення* процесу. Саме зі стандартним *уведенням* працюють всі операції читання, якщо в них не зазначений дескриптор файлу. Другий (дескриптор 1) - відкритий на **запис**, це *стандартний вивід* процесу. З ним працюють всі операції запису, якщо дескриптор файлу не зазначений у них явно. Нарешті, третій потік даних (дескриптор 2) призначається для *виводу діагностичних повідомлень*, він називається ***стандартний вивід помилок***. Оскільки ці три дескриптори вже відкриті до моменту запуску процесу, перший файл, відкритий **самим** процесом, буде, швидше за все, мати дескриптор 3.

Дескриптор- це описатель потоку даних, відкритого процесом. Дескриптори нумеруються, починаючи з 0. При відкритті нового потоку даних його дескриптор одержує найменший з невикористовуваних у цей момент номерів. Три заздалегідь відкритих дескриптори - стандартне *уведення* (0), *стандартний вивід* (1) і *стандартний вивід помилок* (2) - видаються при запуску.

Коли запускається *стартовий командний інтерпретатор* , всі три заздалегідь відкритих дескриптори зв'язані в нього з *терміналом* (точніше, з відповідним файлом-*діркою* типу tty): користувач уводить команди із клавіатури й бачить повідомлення на екрані. Отже, будь- яка команда, що запускається з командної оболонки, буде виводити на той же *термінал*, а будь-яка команда, запущена **интерактивно** (не в тлі) - уводити звідти.

Стандартний вивід

Одним зі способів зберегти *вивід* програми у файлі замість того, щоб виводити його на монітор: поставити знак ">" і вказати після нього ім'я файлу:

```
[methody@localhost methody]$ cat > textfile Це файл
для прикладів.
^D
```

Від використання *символу* ">" можливості самої утиліти cat, звичайно, не розширилися. Більше того, cat у цьому прикладі не одержала від **командної оболонки** ніяких параметрів: ні знака ">", ні наступного ім'я файлу. У цьому випадку cat працювала як звичайно, не знаючи (і навіть не цікавлячись!), куди потраплять виведені дані: на екран монітора, у файл або куди-небудь ще. Замість того, щоб самої забезпечувати доставку *виводу* до кінцевого адресата (будь те людин або файл), cat відправляє всі дані на *стандартний вивід* (скорочено - stdout).

Підміна *стандартного виводу* - завдання командної оболонки (shell) . У даному прикладі shell створює порожній файл, ім'я якого зазначено після знака ">", і дескриптор цього файлу передається програмі cat під номером 1 (*стандартний вивід*). Робиться це дуже просто. У лекції 5 було розказано про те, як запускаються команди з оболонки. Зокрема, після виконання fork()

з'являється два однакових процеси, один із яких - дочірній - повинен запустити замість себе команду (виконати exec()). Отож, перед цим він **закриває стандартний вивід** (дескриптор 1 звільняється) і **відкриває** файл (з ним зв'язується перший вільний дескриптор, тобто 1), а запускається команде, що, нічого знати й не треба: її *стандартний вивід* уже підмінений. Ця операція називається **перенапрямком стандартного виводу**. У тому випадку, якщо файл уже існує, shell запише його заново, повністю знищивши все, що в ньому втримувалося до цього. Тому
Методію, щоб продовжити записувати дані в textfile, буде потрібно інша операція - ">>":

[methody@localhost methody]$ cat >> textfile Приклад 1.
^D
[methody@localhost methody]$ cat textfile Це файл для прикладів.
Приклад 1. [methody@localhost methody]$

Стандартный вивід (standard output, stdout) - це потік даних, що відкривається системою для кожного процесу в момент його запуску й призначений для даних, виведених процесом.

Стандартне уведення

Аналогічним образом для передачі даних на вхід програмі може бути використане *стандартне уведення* (скорочено - stdin). При роботі з командним рядком *стандартне введення* - це *символи*, що вводяться користувачем із клавіатури. *Стандартне уведення* можна **перенаправляти** за допомогою командної оболонки, подавши на нього дані з деякого файлу. *Символ* "<" служить для перенапрямку вмісту файлу на *стандартне уведення* програмі. Наприклад, якщо викликати утиліту sort без параметра, вона буде читати рядка зі *стандартного уведення*. Команда "sort < ім'я_файлу" подасть на *уведення* sort дані з файлу:

[methody@localhost methody]$ sort < textfile Приклад 1.
Це файл для прикладів.
[methody@localhost methody]$

Результат дії цієї команди аналогічний команді sort textfile - різниця лише в тім, що коли використовується "<", sort одержує дані зі *стандартного уведення* нічого не знаючи про файл "textfile", звідки вони надходять. Механізм роботи shell у цьому випадку той же, що й при перенапрямку *виводу*: shell читає дані з файлу "textfile", запускає утиліту sort і передає їй на *стандартне уведення* вміст файлу.

Необхідно пам'ятати, що операція ">" **деструктивна** : вона завжди створює файл нульової довжини. Тому для, допустимо, сортування даних **у**

файлі треба застосовувати послідовно sort < файл > новий_файл і mv новий_файл файл. Команда виду команда < файл > той_же_файл просто уріже його до нульової довжини!

Стандартне уведення (standard input, stdin) - потік даних, що відкривається системою для кожного процесу в момент його запуску й призначений для *уведення* даних.

Стандартний вивід помилок

Стандартний вивід помилок (standard error, stderr) - потік даних, що відкривається системою для кожного процесу в момент його запуску й призначений для
діагностичних повідомлень, виведених процесом.

Використання *стандартного виводу помилок* поряд зі *стандартним виводом*
дозволяє відокремити властиво результат роботи програми від різноманітної супровідної інформації, наприклад, направивши їх у різні файли. *Стандартний вивід помилок* може бути переспрямований так само, як і *стандартне уведення*/*вивід*, для цього використовується комбінація *символів* "2>":

Цього разу на *термінал* уже нічого не потрапило, *стандартний вивід* відправився у файл cat.info, *стандартний вивід помилок* - в cat.stderr. Замість ">" і "2>" можна написати "1>" і "2>". Цифри в цьому випадку позначають номери дескрипторів файлів, що відкриваються. Якщо якась утиліта очікує одержати **відкритий** дескриптор з номером, допустимо, 4, те, для того щоб неї запустити, **обов'язково** буде потрібно використовувати сполучення "4>".
Іноді, однак, потрібно об'єднати *стандартний вивід* і *стандартний вивід помилок* в одному файлі, а не розділяти їх. У командній оболонці bash для цього є спеціальна послідовність "2>&1". Це означає "направити *стандартний вивід помилок* туди ж, куди й *стандартний вивід*":

Перенапрямок у нікуди

Іноді свідомо відомо, що якісь дані, виведені програмою, не знадобляться. Наприклад, попередження зі *стандартного виводу помилок*. У цьому випадку можна перенаправляти *стандартний вивід помилок* у файл-*дірку*, спеціально призначений для знищення даних - /dev/null. Усе, що записується в цей файл, просто буде викинуте й **ніде не збережеться**:

Точно в такий же спосіб можна позбутися й від *стандартного виводу*, відправивши його в /dev/null.

Обробка даних у потоці

Конвеєр

Нерідко виникають ситуації, коли потрібно обробити *вивід* однієї програми якоюсь іншою програмою. Користуючись перенапрямком уведення-*виводу*, можна зберегти *вивід* однієї програми у файлі, а потім направити цей файл на *уведення* іншій програмі. Однак те ж саме можна зробити й більш ефективно: перенаправляти *вивід* можна не тільки у файл, але й **безпосередньо** на *стандартне уведення* іншій програмі. У цьому випадку замість двох команд

буде потрібно тільки одна - програми передають один одному дані "з рук у руки". В Linux такий спосіб передачі даних називається ***конвеєр***.

В bash для перенапрямку *стандартного виводу* на *стандартне уведення* іншій програмі служить *символ* "|". Найпростіший і найпоширеніший випадок, коли потрібно використовувати *конвеєр*, виникає, якщо *вивід* програми не вміщається на екрані монітора й дуже швидко "пролітає" перед очами, так що людина не встигає його прочитати. У цьому випадку можна направити *вивід* у програму перегляду (less), що
дозволить не кваплячись пролистать весь текст, повернутися до початку й т.п.:

[methody@localhost methody]$ cat cat.info | less

Можна послідовно обробити дані декількома різними програмами, перенаправляючи *вивід* на *уведення* наступній програмі й організувати як завгодно довгий *конвеєр* для обробки даних. У результаті виходять дуже довгі командні рядки виду "cmd1 | cmd2 | ... | cmd", які можуть здатися громіздк і незручними, але виявляються дуже корисними й ефективними при обробці великої кількості інформації, як ми побачимо далі в цій лекції.

Організація *конвеєра* влаштована в shell по тій же схемі, що й перенапрямок у файл, але з використанням особливого об'єкта системи - *каналу*. Якщо файл можна представити у вигляді коробки з даними, постаченої клапаном для читання або клапаном для запису, то *канал* - це обидва клапани, приклеєні друг до друга взагалі без коробки. Для визначеності між клапанами можна представити трубу, що негайно доставляє дані від входу до виходу (англійський термін - "pipe" - заснований саме на цьому поданні, а в ролі труби виступає, звичайно ж, сам Linux). *Каналом* користуються відразу два процеси: один пише туди, іншої читає. Зв'язуючи дві команди *конвеєром*, shell відкриває *канал* (заводиться **два** дескриптори - вхідн і вихідний), підмінює по вже описаному алгоритмі *стандартний вивід* першого процесу на вхідний дескриптор *каналу*, а *стандартне уведення* другого процесу - на вихідний дескриптор *каналу*. Після чого залишається запустити по команді в цих процесах, і *стандартний вивід* першої потрапить на *стандартне уведення* другої.

Канал (pipe) - неподільна пара дескрипторів (вхідний і вихідний), зв'язаних один з одним таким чином, що дані, записані у вхідний дескриптор, будуть негайно доступні на читання з вихідного дескриптора.

Фільтри

Якщо програма й уводить дані, і виводить, то її можна розглядати як трубу, у яку щось входить і з якої щось виходить. Звичайно зміст роботи таких програм полягає в тім, щоб певним чином **обробити** дані, що надійшли,. В Linux такі програми називають ***фільтрами*** : дані проходять через них, причому щось "застряє" у *фільтрі* й не з'являється на виході, а щось змінюється, щось проходить крізь *фільтр* незмінним. *Фільтри* в Linux звичайно за замовчуванням читають дані зі *стандартного уведення*, а виводять на *стандартний вивід*. Найпростішим *фільтром* Методій уже користувався багато разів - це програма cat: властиво, ніякої "фільтрації" даних вона не робить, вона просто копіює *стандартне уведення* на *стандартний вивід*.

Дані, що проходять через *фільтр* , являють собою текст: у стандартних потоках уведення-*виводу* всі дані передаються у вигляді *символів*, рядок за рядком, як і в *терміналі*. Тому можуть бути состыкованы за допомогою *конвеєра уведення* й *вивід* будь-яких двох програм, що підтримують стандартні потоки *вводу-виводу*. Це нагадує конструктор, всі деталі якого сполучаються між собою.

Принцип комбінування елементарних операцій для виконання складних завдань успадкований Linux від операційної системи UNIX (як і багато інших принципів). Переважна більшість утиліт UNIX не втратили свого значення й в Linux. Всі вони орієнтовані на роботу з даними в текстовій формі, багато хто є *фільтрами*, усе не мають графічного інтерфейсу й викликаються з командного рядка. Цей пакет утиліт називається coreutils.

Структурні одиниці тексту

Роботу в системі Linux майже завжди можна представити як роботу з текстами. Пошук файлів і інших об'єктів системи - це одержання від системи тексту **особливої** структури - списку імен. Операції над файлами - створення, перейменування, переміщення, а також сортування, перекодування та інше - заміну одних *символів* і рядків іншими або в каталогах, або в самих файлах. Настроювання системи в Linux зводиться безпосередньо до роботи з текстами - редагуванню *конфігураційних файлів* і написанню *сценаріїв* (докладніше про це див. лекції 8 і 12).

Працюючи з текстом в Linux, потрібно брати до уваги, що текстові дані, передані в системі, структуровані. Більшість утиліт обробляє не безперервний потік тексту, а послідовність **одиниць**. У текстових даних в Linux виділяються наступні структурні одиниці:

1 **Рядок**

Рядок - основна одиниця передачі тексту в Linux. *Термінал* передає дані від користувача системі рядками (командний рядок), безліч утиліт уводять і виводять дані построчно, при роботі багатьох утиліт одному рядку відповідає один об'єкт системи (ім'я файлу, шлях і т.п.), sort сортує рядка. Рядки розділяються *символом* кінця рядка "\n" (newline).

2 **Поле**

В одному рядку може згадуватися й більше одного об'єкта. Якщо розуміти об'єкт як послідовність *символів* з певного набору (наприклад, букв), то рядок можна розглядати як складається зі слів і роздільників. У цьому випадку текст від початку рядка до першого *роздільника* - це перше *поле*, від першого *роздільника* до другого - друге *поле* й т.д. Як *роздільник* можна розглядати будь-який *символ*, що не може використовуватися в об'єкті. Наприклад, якщо в шляху "/home/methody" *роздільником* є *символ* "/", те перше *поле* порожньо, друге містить слово "home", третє - "methody". Деякі утиліти дозволяють вибирати з рядків окремі *поля* (по номері) і працювати з рядками як з таблицею.

3 **Символи**

Мінімальна одиниця тексту - *символ*. ***Символ*** - це одна буква або інший письмовий знак. Стандартні утиліти Linux дозволяють заміняти одні *символи*

іншими (робити транслітерацію), шукати й заміняти в рядках *символи* й комбінації *символів*.

Символ кінця рядка в кодуванні ASCII збігається з *керуючою послідовністю* "^J" - "переклад рядка", однак в інших кодуваннях він може бути іншим. Крім того, на більшості *терміналів* - але не на всіх! - слідом за перекладом рядка необхідно виводити ще *символ* повернення каретки ("^M"). Це викликало плутанину: деякі системи вимагають, щоб наприкінці текстового файлу стояли **обоє цих символу** в певному порядку. Щоб уникнути плутанини, в UNIX (і, як наслідок, в Linux) було прийнято єдино вірне рішення: уміст файлу відповідає кодуванню, а при *виводі* на *термінал* кінці рядка перетворяться в *керуючі послідовності* відповідно до настроювання *термінала*.

У розпорядженні користувача Linux є ряд утиліт, що виконують елементарні операції з одиницями тексту: пошук, заміну, поділ і об'єднання рядків, *полів*, *символів*. Ці утиліти, як правило, мають однакове подання про те, як визначаються одиниці тексту: що таке рядок, які *символи* є *роздільниками* й т.п. У багатьох випадках їхнього подання можна змінювати за допомогою настроювань. Тому такі утиліти легко взаємодіють один з одним. Комбінуючи їх, можна автоматизувати досить складні операції по обробці тексту.

Наприклад, стандартна утиліта для підрахунку рядків, слів і *символів* - wc (від англ. "word count" - "підрахунок слів"). Підрахувати кількість файлів в каталозі можна в наступний спосіб:

[methody@localhost methody]$ find . | wc -l 42

Задавши find критерії пошуку, можна порахувати й що-небудь менш тривіальне, наприклад, файли, які створювалися або були змінені в певний проміжок часу, файли з певним режимом доступу, з певним ім'ям і т.п. Довідатися про всі можливості пошуку за допомогою find і підрахунку за допомогою wc можна з руководств по цих програмах.

Пошук

Найчастіше користувачеві потрібно знайти тільки згадування чогось конкретного серед даних, виведених утилітою. Звичайно це завдання зводиться до пошуку рядків, у яких зустрічається певне слово або комбінація *символів*. Для цього підходить стандартна утиліта grep. grep може шукати рядок у файлах, а може працювати як *фільтр*: одержавши рядка зі *стандартного уведення*, вона виведе на *стандартний вивід* тільки ті рядки, де зустрілося шукане сполучення *символів*.

Пошук по регулярному вираженню

Дуже часто точно не відомо, яку саме комбінацію *символів* потрібно буде знайти. Точніше, відомо тільки те, як приблизно повинне виглядати шукане слово, що в нього повинне входити й у якому порядку. Так звичайно буває, якщо деякі фрагменти тексту мають строго певний формат. Наприклад, у руководствах, виведених програмою info, прийнятий такий формат посилань: "*Note назва_вузла::". У цьому випадку потрібно шукати не конкретне сполучення *символів*, а "Рядок "*Note", за якої треба назва вузла (одне або кілька слів і пробілів), що кінчається *символами* "::"". Комп'ютер цілком

здатний виконати такий запит, якщо його сформулювати на строгому й зрозумілому йому мові, наприклад, мовою *регулярних виражень*. ***Регулярне вираження*** - це спосіб однією формулою задати всі послідовності *символів*, що підходять користувачеві:

У *регулярному вираженні* більшість *символів* позначають самі себе, як якби ми шукали звичайний текстовий рядок, наприклад, "Note" і "::" у *регулярному вираженні* відповідають рядкам "Note" і "::" у тексті. Однак деякі *символи* мають спеціальне значення, самий головний з таких *символів* - зірочка ("*"), поставлена після елемента *регулярного вираження*, позначає, що можуть бути знайдені тексти, де цей елемент повторений будь-яка кількість разів, у тому числі й жодного, тобто просто відсутній.

Мовою *регулярних виражень* можна також позначити "будь-який *символ*" ("."), "одне або більше збігів" ("+"), початок і кінець рядка ("^" і "$" відповідно) і т.д. Завдяки *регулярним вираженням* можна автоматизувати дуже багато завдань, які в противному випадку зажадали б величезної й кропіткої роботи людини. Більше докладні відомості про можливості мови *регулярних виражень* можна одержати з керівництва regex(7).

Регулярні вираження в Linux використовуються не тільки для пошуку програмою grep. Дуже багато програм, так чи інакше працюючі з текстом, у першу чергу текстові редактори, підтримують *регулярні вираження*. До таких програм ставляться два "головних" текстових редактори Linux - Vim і Emacs, про які мова йтиме в наступній лекції
(9). Однак потрібно враховувати, що в різних програмах використовуються різні діалекти мови *регулярних виражень*, де ті самі поняття мають різні позначення, тому завжди потрібно звертатися до посібника з конкретної програми.

На закінчення можна сказати, що *регулярні вираження* дозволяють різко підвищити ефективність роботи, добре інтегровані в робітниче середовище в системі Linux, і їсти зміст витратити час на їхнє вивчення.

Заміни

Зручність роботи з потоком не в останню чергу полягає в тому, що можна не тільки вибірково передавати результати роботи програм, але й автоматично заміняти один текст іншим прямо в потоці.

Для заміни одних *символів* іншими призначена утиліта tr (скорочення від англ. "translate" - "перетворювати, переводити"), що працює як *фільтр*. Методій вирішив застосувати її прямо по призначенню й виконати при її допомозі транслітерацію - заміну латинських *символів* близькими по звучанню росіянами:

Крім простої заміни окремих *символів*, можлива заміна послідовностей (слів). Спеціально для цього призначений потоковий редактор sed (скорочення від англ. "stream editor"). Він працює як *фільтр* і виконує редагування вступників рядків: заміну одних послідовностей *символів* іншими, причому можна заміняти й *регулярні вираження*.

В sed дуже широкі можливості, але досить незвичний синтаксис, наприклад, заміна виконується командою "s/що_ заміняти/на_що_заміняти/".

Щоб у ньому розібратися, потрібно обов'язково прочитати керівництво sed(1) і знати *регулярні вираження*.

Основи

Системний виклик fork(), створюючи точну копію батьківського процесу, копіює також і оточення. Необхідність в "оточенні" обумовлена от яким завданням. Акт передачі даних від батьківського процесу дочірньому, і, що ще важливіше, системі, повинен мати властивість атомарности. Якщо використовувати для цієї мети файл (наприклад, конфігураційний файл запускається программы, що), завжди зберігається ймовірність, що за час між зміною файлу й наступним читанням запущеною програмою хтось - наприклад, інший процес того ж користувача - знову змінить цей файл. Добре б, щоб зміна даних і їхня передача виконувалися однією операцією. Наприклад, завести для кожного процесу такий "файл", уміст якого, по-перше, міг би змінити тільки цей процес, і, по-друге, воно автоматично копіювалося б в аналогічний "файл" дочірнього процесу при його породженні.

Ці властивості й реалізовані в понятті "оточення". Кожний процесс, що запускається, система постачає якимось інформаційним простором, що цей процес вправі змінювати як йому заманеться. Правила користування цим простором прості: у ньому можна задавати іменовані сховища даних (змінні оточення), у які записувати яку завгодно інформацію (привласнювати значення змінної оточення), а згодом цю інформацію зчитувати (підставляти значення змінної). дочірній процес - точна копія батьківського, тому його оточення - також точна копія батьківського. Якщо про дочірній процес відомо, що він використовує значення деяких змінних із числа переданих йому з оточенням, батьківський може заздалегідь указати, яким з копируемых в оточенні змінних потрібно змінити значення. При цьому, з одного боку, ніхто (крім системи, звичайно) не зможе втрутитися в процес передачі даних, а з іншого боку, та сама утиліта може бути
використана тим самим способом, але в зміненому оточенні - і видавати різні результати:

[methody@localhost methody]$ date неділя, 9
листопада 2008 10:37:38 +0200
[methody@localhost methody]$ LC_TIME=C; date Sun
Nov 9 10:38:12 EET 2008

Оточення (environment) - набір даних, приписаних системою процесу. Процес може користуватися інформацією з оточення для настроювання, змінювати й доповнювати його. Оточення представлене у вигляді змінні оточення і їхні значення. При породженні процесу оточення батьківського процесу успадковується дочірнім (копіюється).

Робота зі змінними в shell

Якщо розглядати shell у якості високорівневої мови програмування, то його змінні - самі звичайні строкові змінні. Записати значення в змінну можна за

допомогою операції присвоювання, а прочитати його звідти - за допомогою операції підстановки виду $змінна:

```
[methody@localhost methody]$ A=dit
[methody@localhost methody]$ C=dah
[methody@localhost methody]$ echo $A $B $C dit
dah
[methody@localhost methody]$ B=" "
[methody@localhost methody]$ echo $A $B $C dit
dah
[methody@localhost methody]$ echo "$A $B $C" dit
  dah
[methody@localhost methody]$ echo '$A $B $C'

  $A $B $C
```

Як видно із приклада, значення невизначеної змінної (B) в shell уважається порожнім і при підстановці не виводиться ніяких попереджень. Сама підстановка відбувається, як і генерація імен, перед розбором командного рядка, набраної користувачем. Тому друга команда echo у прикладі одержала, як і перша, два параметри ("dit" і "dah"), незважаючи на те, що змінна B була на той час визначена й містила роздільник-пробіл. А от третя й четверта команди echo одержали по одному параметрі. Тут позначилося розходження між одинарними й подвійними лапками в shell: усередині подвійних лапок діють підстановка значень змінних.

Змінні, які командний інтерпретатор bash визначає після запуску, не належать оточенню, і, стало бути, не успадковуються дочірніми процесами. Щоб змінна bash
потрапила в оточення, її треба експортувати командою export:

```
[methody@localhost methody]$ echo "$Qwe ---i $LANG"
  ---i ru_RU.KOI 8-R
[methody@localhost methody]$ Qwe="Rty" LANG=C
[methody@localhost methody]$ echo "$Qwe ---i $LANG" Rty ---i
C
[methody@localhost methody]$ sh sh-
2.05b$ echo "$Qwe ---i $LANG"
  ---i C sh-
2.05b$ exit
[methody@localhost methody]$ echo "$Qwe ---i $LANG" Rty ---i
  C
[methody@localhost methody]$ export Qwe
[methody@localhost methody]$ sh sh-2.05b$ echo
"$Qwe ---i $LANG"
```

Rty ---i C sh-

2.05b$ exit

В цьому прикладі створюється нова змінна Qwe і змінюється значення змінної оточення LANG, що дісталася стартовому bash від програми login. У результаті запущений дочірній процес sh одержав змінене значення LANG і ніякий змінної Qwe в оточенні. Після export Qwe ця змінна була додана в оточення й, відповідно, передалася sh.

Змінні оточення, що використовуються системою й командним інтерпретатором

Під час сеансу роботи користувача стартовий командний інтерпретатор одержує від login досить багате оточення, до якого додає й власні настроювання. Переглянути оточення в bash можна за допомогою команди set. Більшість заздалегідь певних змінних використовуються або самою командною оболонкою, або утилітами системи, тому їхня зміна приводить до того, що оболонка або утиліти починають працювати трохи інакше.

Досить примітна змінна оточення PATH. У ній утримується список каталогів, елементи якого розділяються двокрапками. Якщо команда в командному рядку - не власна команда shell (начебто cd) і не представлена у вигляді шляху до запускається файлу, що (як /bin/ls або ./script), те shell буде шукати цю команду серед імен файлів, що запускаються, у всіх каталогах PATH, і тільки в них. Точно так само будуть надходити й інші утиліти, що використовують бібліотечну функцію execlp() або execvp() (запуск програми).

Із цієї причини виконуються файлы, що, неможливо запускати просто по імені, якщо вони лежать у поточному каталозі, і поточний каталог не входить в PATH. В таких випадках можна користуватись найкоротшим з можливих шляхів, "./" (наприклад, викликаючи сценарій ./script):

Змінні оточення, що впливають на роботу різних утиліт, досить багато. Наприклад, змінні сімейства LC_(повний їхній список видається командою locale), що визначають мову, на якому виводяться діагностичні повідомлення, стандарти на формат дати, грошових одиниць, чисел, способи перетворення рядків і т.п. Дуже важлива змінна TERM, що визначає тип термінала: як відомо з лекції 2, різні термінали мають різні керуючі послідовності, тому програми, що бажають ці послідовності використовувати, обов'язково звіряються зі змінної TERM. Якщо якась утиліта вимагає редагування файлу, цей файл передається програмі, шлях до якої зберігається в змінної EDITOR (звичайно це /usr/bin/vi. Нарешті, деякі змінні, наприклад, UID, USER або PWD просто містять корисну інформацію, яку можна було б добути й іншими способами.

Деякі змінні оточення призначені спеціально для bash: вони задають його властивості й особливості поводження. Такі змінні сімейства PS (Prompt String). У цих змінні зберігається рядок-підказка, що командний інтерпретатор

виводить у різних станах. Зокрема, уміст PS1 - це підказка, що shell показує, коли вводить командний рядок, а PS2 - коли користувач натискає Enter, а інтерпретатор з якоїсь причини вважає, що введення командного рядка ще незавершено (наприклад, не закриті лапки).

Вміст PS1 не просто підставляється при виводі - воно ще й перетвориться, дозволяючи виводити всяку корисну інформацію: ім'я користувача (відповідає подстроке "\u", user), ім'я комп'ютера ("\h", host), час ("\t", time), шлях до поточного каталогу ("\w", work directory) і т.п. Таке перетворення значень змінні сімейства PS1 виконується, тільки
коли їх використовує bash як підказка, а при звичайній підстановці цього не відбувається:

```
[methody@localhost methody]$ cd examples/
[methody@localhost examples]$ echo $PS1 [\u@\h
\W]\$

[methody@localhost examples]$ PS1=" ---i> "

---i>

---i>
PS1="\t  \w
"  22:11:47
~ 22:11:48 ~

22:11:48  ~  PS1="\u@\h:\w  \$  "
methody@localhost:~/examples  $
methody@localhost:~/examples  $
methody@localhost:~/examples  $  cd
methody@localhost:~ $
```

Скриптова мова програмування sh

Більша частина того, що потрібно починаючому користувачеві Linux, робиться за допомогою однієї правильної команди, або викликом декількох команд у конвеєрі. Від користувача тільки потрібно оформити рішення завдання у вигляді сценарію на shell. Насправді ж уже найперший з командних інтерпретаторів, sh, був високорівневою мовою програмування - якщо, звичайно, уважати всі утиліти системи його операторами. При такому підході від sh потрібно зовсім небагато: можливість викликати утиліти, можливість вільно маніпулювати результатом їхньої роботи й кілька алгоритмічних конструкцій (умови й цикли).

Писати сценарії для bash - непрактично, тому що виконуватися вони зможуть лише за допомогою bash. Якщо ж обмежити себе рамками sh, сумісність із яким

оголошена й в bash, і в zsh, і в ash (найбільш близькому по можливостях до sh), і в інших командних інтерпретаторах, виконуватися ці сценарії зможуть кожним з sh-подібних інтерпретаторів, і не тільки в Linux.

Інтеграція процесів

Кожний процес Linux при завершенні передає батьківському код повернення (exit status), що дорівнює нулю, якщо процес уважає, що його робота була успішної, або номеру помилки - у противному випадку. Командний інтерпретатор зберігає код повернення останньої команди в спеціальної змінної "?". Що більше важливо, код повернення використовується в умовних операторах: якщо він дорівнює нулю, умова вважається виконаною, а якщо немає — невиконаною.

Великою кількістю функцій володіє команда test: вона вміє порівнювати числа й рядки, перевіряти ярлик об'єкта файлової системи й наявність самого цього об'єкта. В "test" є друге ім'я: "[" (як правило, /usr/bin/[- символьна або навіть тверде посилання на /usr/bin/test), що дозволяє оформляти оператор if більше звичним образом.

Другий тип підстановки, що shell робить усередині подвійних лапок - це підстановки виводу команди. Підстановка виводу має вигляд "`команда`", тобто команда вводиться між лапками слабкого наголосу, (інший варіант - "$(команда)"). Як і підстановка значення змінної, вона відбувається перед тим, як почнеться розбір командного рядка: виконавши команду й одержавши від її якийсь текст, shell прийметься розбирати його, як якби цей текст користувач набрав вручну. Це дуже зручний засіб, якщо те, що виводить команда, необхідно передати самому інтерпретаторові:

```
[methody@localhost methody]$ A=8; B=6
[methody@localhost methody]$ expr $A + $B 14

[methody@localhost methody]$ echo "$A + $B = 'expr $A + $B'"
8 + 6 = 14

[methody@localhost methody]$ A=3.1415; B=2.718

[methody@localhost methody]$ echo "$A + $B = 'expr $A + $B'"

expr: нечисловий аргумент
3.1415 + 2.718 =

[methody@localhost methody]$ echo "$A + $B" | bc 5.8595
```

[methody@localhost methody]$ C='echo "$A + $B" | bc'
[methody@localhost methody]$ echo "$A + $B = $C" 3.1415 + 2.718 = 5.8595

Стартові сценарії

Настроювання оболонки - це в першу чергу настроювання оточення. На початку сеансу роботи (при запуску стартового командного інтерпретатора) за допомогою команди "." виконується сценарій з файлу зі спеціальним ім'ям - /etc/profile. Це - так званий загальносистемний профіль, стартовий сценарій, що виконується при вході в систему кожного, хто використовує командну оболонку, подібну sh. Слідом виконується персональний профіль (або просто профіль) користувача - сценарій, що перебуває в домашньому каталозі, і що називається .profile. Цей сценарій користувач може видозмінювати, як йому заманеться. Що стосується bash, те структура його стартових файлів складніше. Насамперед, ~/.profile виконується, тільки якщо в домашньому каталозі немає файлу .bash_profile або .bash_login, інакше стартовий сценарій береться звідти. У ці файли можна поміщати команди, несумісні з іншими версіями shell, наприклад, керування скороченнями або прив'язку функцій до клавіш. Крім того, кожний інтерактивний (взаємодіючий з користувачем), але не стартовий bash виконує системні й персональний конфігураційні сценарії /etc/bashrc і ~/.bashrc. Щоб стартовий bash також виконував ~/.bashrc, що відповідає команду необхідно вписати в ~/.bash_profile. Далі, кожний неінтерактивний (запущений для виконання сценарію) bash звіряється зі змінної оточення BASH_ENV і, якщо в цій змінній записане ім'я існуючого файлу, виконує команди звідти. Нарешті, при завершенні стартового bash виконуються команди з файлу ~/.bash_logout.