МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Український державний морський технічний університет
імені адмірала Макарова

А.Ю. Гайда, О.Б. Димо

Інструментальні засоби операційних систем UNIX

Частина I

Рекомендовано Методичною радою УДМТУ
як методичні вказівки

Миколаїв 2003

УДК 004.451(076)

Гайда А.Ю., Димо О.Б. Інструментальні засоби операційних систем UNIX: Методичні вказівки. – Миколаїв: УДМТУ, 2003. – 44 с.

Кафедра інформаційних технологій

Викладені основи використання інструментальних засобів розробки програм в операційних системах UNIX/Linux. Розглянуті питання інструментального забезпечення програмного проекту; мови програмування Tcl, Python; засоби взаємодії процесів у локальній мережі та мережі Internet. Акцентовано увагу на використанні мобільних інструментальних засобів.

Призначено для студентів напряму "Комп'ютерні науки".

Рецензенти: д-р техн. наук, проф. Кошкін К.В.;
д-р техн. наук, проф. Казарезов О.Я.

ВСТУП

Операційні системи (ОС) UNIX і Linux використовуються в галузях, що вимагають забезпечення таких характеристик інформаційних систем, як готовність – зберігання даних таким чином, що дозволяє надійно отримати їх своєчасно за наявності відповідних прав; цілісність – забезпечення достовірності даних і їх внутрішньої повноти й непротиріччя; конфіденційність – забезпечення кількох рівнів захисту інформаційних ресурсів від несанкціонованого використання. Можливості ОС UNIX і Linux дозволяють застосовувати ці ОС на всіх рівнях інформаційних систем – від реалізації серверних рішень до автоматизованих робочих місць. Використання UNIX і Linux у сфері серверних обчислень традиційне.

У посібнику розглянуті питання використання інструментальних засобів розробки програмного забезпечення в ОС UNIX і Linux: засоби розробки програмного забезпечення поширеними мовами програмування C/C++, Tcl, Python; взаємодія програм у мережі за допомогою програмних гнізд.

Приклади програм виконані та перевірені в ОС Red Hat Linux (версії 7.2, 7.3, 8.0), використані компілятор 'gcc' версії 2.96 та 3.2, інтерпретатори 'Tcl' (версія 8.3), 'Python' (версії 1.5, 2.1 та 2.2). Для набору і форматування тексту застосовано пакет 'Open Office' (версії 638, 641, 1.0).

ПОЗНАЧЕННЯ В ТЕКСТІ І ШРИФТОВІ ВИДІЛЕННЯ

моноширинний шрифт – тексти програм, команди в консолі;

моноширинний шрифт, курсив – тексти повідомлень програм;

'текст у лапках' – назви файлів, проектів, змінних тощо;

<текст у кутових дужках> – позначення аргументів, що потребують заміни;

[текст у прямокутних дужках] – необов'язкові аргументи.

1. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РОЗРОБКИ ПРОГРАМ

1.1. Засоби побудови програмного проекту (make, autoconf, automake)

Засоби побудови програмного проекту використовуються для автоматизації процесу складання проекту з урахуванням специфічних особливостей системи користувача, таких, як тип ОС, наявність відповідних стандартних заголовків або бібліотек тощо.

Утиліта make дозволяє автоматизувати виконання визначених над файлами дій для досягнення вказаних цілей з урахуванням певних залежностей. Виконуються тільки ті дії, для яких відсутні або застаріли результати обробки файлів. Загалом використовується для компіляції і складання програмних проектів (як правило, вихідний файл програми знаходиться у залежності від об'єктних файлів, які, в свою чергу, створюються в результаті компіляції відповідних файлів з вихідними текстами).

Опис залежностей файлів, визначення дій і цілей виконуються в спеціальному текстовому файлі ('*Makefile*' або '*makefile*'), на його основі *make* визначає вплив модифікацій одних файлів на інші, точну послідовність дій, необхідних для створення нової версії, і команди, які потрібно виконати для досягнення вказаної мети. *Makefile* може бути створений в текстовому редакторі, згенерований засобами розробки програм або отримано з шаблона.

Для отримання файла з шаблона використовується команда *configure*, яка генерується автоматично разом з шаблоном багатьма IDE. Призначення *configure* – внести зміни в *Makefile* з метою врахування індивідуальних особливостей конфігурації конкретної ОС і платформи.

Простий *Makefile* складається з "правил" (*rules*) наступним чином:

ціль ... : пререквизит ...
    команда
    ...

Правило (*rule*) вказує, коли і яким чином слід поновлювати файли, вказані в ньому, в якості цілі.

Ім'я цілі (*target*) може представляти собою ім'я файла, який генерується в процесі роботи утиліти *make* (наприклад, об'єктні та командні файли), або ім'я дії, що треба виконати (наприклад, '*clean*' – очистити).

Пререквизит (*prerequisite*) – це файл, який використовується як вхідні дані для породження цілі. Ціль може бути в залежності відразу від кількох файлів, які перераховують через пробіл.

Команда – це дія, що виконується утилітою '*make*' над пререквизитом для досягнення цілі. Правило може включати кілька команд – кожна окремим рядком. Команди виконуються в командному інтерпретаторі (*shell*). Рядки з командами обов'язково повинні починатися з символа табуляції.

Приклад. Програма '*hello*' може бути отримана з вхідних файлів 'x.c', 'y.c', 'z.c' шляхом їх компіляції і редагування зв'язків разом з бібліотекою '*libm.a*'. Файли 'x.c' і 'y.c' використовують описи з файла '*defs.h*', а 'z.c' – не використовує.

Залежності й команди можуть бути записані таким чином:

```
hello : x.o y.o z.o
    cc x.o y.o z.o -lm -o hello
x.o : x.c  defs.h
    cc -c x.c
y.o : y.c defs.h
    cc -c y.c
z.o : z.c
    cc -c z.c
```

Якщо цю інформацію розмістити в файлі з ім'ям '*makefile*', команда `make` виконає операції, необхідні для перегенерації '*hello*' після змін у будь-якому з вхідних файлів 'x.c', 'y.c', 'z.c', '*defs.h*'.

Опції командного рядка *make*:

`-f <filename>`: вказує ім'я файла проекта (якщо опція не використовується, береться '*Makefile*' або '*makefile*');

`-k`: вказує продовжувати обробку залежностей цілей навіть тоді, коли при обробці однієї з них виникла помилка (за умовчанням процес зупиняється, як тільки виникла помилка при обробці цілей);

`-n`: виводити команди, які виконує 'make', але не виконувати їх насправді.

Розробка файла 'Makefile' для великого проекту може бути складною задачею з наступних причин:

визначення залежностей для великого проекту є нетривіальною задачею;

складання файла 'Makefile' проводить розробник програмного пакету, а використовує користувач в місці встановлення програмного пакету.

Автоматичне складання 'Makefile' з урахуванням залежностей і особливостей конфігурації системи виконує пакет програм 'Autoconf'.

Утиліта automake – це інструмент, який дозволяє автоматично або на основі спеціально створених макросів визначити залежності між файлами проекту. Вхідною інформацією для 'automake' є файл 'Makefile.am'. Для прикладу, розглянутому раніше, можуть бути визначені залежності (у файлі 'Makefile.am'):

```
bin_PROGRAMS = hello
hello_SOURCES = x.c y.c z.c defs.h
AUTOMAKE_OPTIONS = foreign
```

Останній рядок дозволяє створити пакет, в якому можуть бути відсутні файли зі стандартними документами: 'NEWS', 'README', 'AUTHORS' і 'ChangeLog'.

Результатом роботи 'automake' є файл 'Makefile.in', що враховує залежності між пакетами, але не враховує особливостей конфігурації системи.

Утиліта autoconf призначена для генерації програми 'configure', яка дозволяє врахувати особливості системи (присутність вхідних файлів і бібліотек у системі, визначити шляхи їх знаходження та опції компіляції). Вхідною інформацією для 'configure' є файл 'Makefile.in', вихідною – файл 'Makefile'.

Вхідною інформацією для 'autoconf' є файл 'configure.ac' або 'configure.in', в якому зазначаються імена макросів та параметри до них. Autoconf при обробці вхідних файлів генерує програми для виконання дій, що визначаються макросами (перевірка при-

сутності необхідних файлів, бібліотек, опції компіляції та компоновки тощо).

Пакет automake та всі бібліотеки, які встановлені в системі, запроваджують готовий набір макросів для використання. Звичайно їх можна знайти в каталозі /usr/share/aclocal.

Наприклад, для програми 'hello':

```
AC_INIT(x.c y.c z.c defs.h)
# Ініціалізуємо скрипт configure. При виконанні
# він перевірить наявність вказаних файлів.

AM_INIT_AUTOMAKE(hello, 1.0)
# Вказуємо назву і версію пакета.

AC_PROG_CC
# Перевіряємо наявність компілятора C, необхідного
# для збирання пакета.

AM_PROG_LIBTOOL
# Вказуємо на необхідність застосування програми
# libtool для компіляції та компоновки необхідних
# динамічних бібліотек.

AC_CHECK_LIBM
# Перевіряємо присутність "C math library",
# визначаємо опції компіляції і компоновки з нею.

AC_OUTPUT(Makefile)
# Вказуємо назву вихідного файла.
```

Використання autoconf, automake і make разом. Після складання файлів 'Makefile.am' і 'configure.ac' розробник повинен виконати програми з пакету 'Autoconf' у наступній послідовності:

```
aclocal
autoconf
automake -a -c
```

При використанні програми 'automake' разом із 'libtool' у каталозі з файлом 'configure.ac' або в '/usr/share/automake' повинен знаходитись файл 'ltmain.sh' (за умовчанням він доступний у '/usr/share/libtool').

Результатами виконання програм є програма 'configure', файл 'Makefile.in' і допоміжні файли.

Для складання проекту 'hello' користувач повинен виконати команди в наступній послідовності:

```
./configure
make
make install
```

1.2. Система управління версіями Concurrent Version System

***Concurrent Version System* (*CVS*)** – система управління паралельними версіями (система контроля версій). CVS – розподілена система за технологією клієнт-сервер. Сервер забезпечує зберігання файлів проекту (репозиторій) і дозволяє клієнтам отримати копії файлів (робочий простір) для їх редагування. CVS основана на парадигмі "блокування-зміна-розблокування" та призначена для таких цілей:

зберігання історії змін (можливо, вхідного коду) файлів. CVS зберігає розбіжності між версіями замість створення окремих файлів для кожної версії, а також журнал з інформацією хто, коли і з якою метою вніс зміни. Це дозволяє відтворити будь-яку версію кожного файла;

підтримки колективної розробки. Програміст за допомогою команд CVS може отримати свіжу копію файлів від CVS (checkout) і "повернути" зміни в CVS по завершенню роботи (checkin);

надання робочого простору. CVS не може використовувати звичне дерево каталогів; програміст працює в дереві каталогів, яке створює CVS. Використання окремих елементів робочого простору може бути обмежене для окремих користувачів.

Налаштування репозиторія. Репозиторій зберігається на сервері CVS. Для його використання адміністратор проекта 'project' повинен:

встановити змінну оточення CVSROOT –

```
CVSROOT=~/project
export CVSROOT
```

створити репозиторій: при встановленій CVSROOT –

```
cvs init
```

або без установки CVSROOT –

```
cvs -d /home/master/project init
```

внести необхідні зміни в конфігурацію проекта, яка зберігається у файлі '$CVSROOT/CVSROOT/config';

додати в репозиторій необхідні каталоги (наприклад, 'dir') –

```
cd dir
cvs import dir vendor v1.1
```

Рядок 'vendor' – мітка розробника;

занести копії файлів проекта в репозиторій (наприклад, 'test.c') –

```
cvs import test.c vendor v1.1
```

Редагування файлів і об'єднання змін. Редагування файлів проводиться в робочому просторі проекту. Для його заповнення необхідно:

отримати від CVS дерево каталогів за допомогою команди 'checkout' (для каталога 'dir') –

```
cd
cvs checkout dir
U dir/Makefile
U dir/test.c
```

Рядок 'U файл' означає, що файл був поновлений. Після того, як CVS створив дерево каталогів, файли, що в ньому знаходяться, можуть бути використані для редагування, компіляції тощо;

синхронізувати версії –

```
cvs update
   cvs update: Updating.
   U Makefile
   RCS file: /master/project/test.c,v
   retrieving revision 1.0
   retrieving revision 1.1
   Merging differences between 1.0 and 1.1 into test.c
   M test.c
```

Рядок 'M файл' означає, що файл був модифікований і має зміни, недосяжні іншим розробникам.

Після виконання команди 'cvs update' CVS поєднує зміни тільки в робочому просторі; репозиторій і робочі простори інших розробників залишаються без змін. Для внесення змін в репозиторій необхідно зафіксувати зміни:

```
cvs commit test.c
```

CVS запускає текстовий редактор для опису змін:

```
Checking in test.c;
/home/master/project/test.c,v  <-- test.c
new revision: 1.1; previous revision: 1.0
```

Для ***додавання файла*** або каталога в проект необхідно створити його в робочому просторі, маркувати його для додавання командою 'cvs add', додати його в репозиторій командою 'cvs commit'.

Для ***видалення файла*** з проекта необхідно видалити його з робочого простору, маркувати його для видалення командою 'cvs rm', видалити його з репозиторія командою 'cvs commit'.

Видалення файла з репозиторія не видаляє історію про цей файл – вона доповнюється записом "не існує". Якщо видалити файл з робочого простору, команда 'cvs update' поновить його останню версію з репозиторія.

Для ***перейменування файла*** необхідно перейменувати його в робочому просторі, виконати 'cvs rm' з старим ім'ям і 'cvs add' з новим.

Історія про старий файл не переноситься на новий файл.

Для ***видалення каталогів*** необхідно видалити всі файли з каталога, вказати опцію '–P' при виконанні 'cvs update' або 'cvs checkout'.

Інформацію про файл надають команди:

про стан файла в репозиторії

```
cvs status <filename>
```

про зміни для файла

```
cvs log <filename>
```

відображення змін для файла

```
cvs diff -c -r 1.0 -r 1.1 <filename>
```

–c – вказує використати зручний формат даних;

–r 1.0 –r 1.1 – вказує вивести зміни з редакції 1.0 до редакції 1.1.

Файл, що може бути отриманий командою 'cvs diff', подібний до результатів команди 'diff' і зветься латкою (patch). Латка може бути використана в команді 'patch' для внесення змін у файли або каталоги з метою переходу від меньшої версії до більшої.

Для ***повернення до попередньої версії*** використовують команди:

для повернення на вказаний термін

```
cvs checkout -D '1 year ago' project
```

для повернення до вказаної версії

```
cvs checkout -r1.0 project
```

Інші можливості CVS. Розглянуті можливості демонструють лише невелику частину всіх можливостей CVS. CVS – складна система, за допомогою якої здійснюється керування значними за розмірами проектами, наприклад, розробка документації, програм тощо. При цьому використовуються такі можливості системи, як взаємодія з віддаленими учасниками проекту, аутентифікація, формування документів до проекту. Найбільш відомий сервер CVS www.sourceforge.net одночасно керує декількома тисячами проектів. Одним з найбільших проектів CVS є ядро ОС Linux (близько 10 млн рядків програмного коду загальним розміром понад 100 Mb).

1.3. GNU Emacs / Xemacs

При розробці застосувань на платформах UNIX та Windows важливим є питання вибору середовища розробки (IDE), яке б дозволило вести цей процес однаково на всіх платформах. Такі засоби, як autoconf, automake, make, cvs, розглянуті в розд. 1.1 і 1.2, дозволяють організувати управління проектом та спільну роботу над ним, але не визначають середовище для ведення процесу розробки.

Найбільш універсальним і водночас функціональним IDE є редактор Emacs (www.gnu.org/software/emacs) або зручніший для використання еквівалент XEmacs (www.xemacs.org). Для розробки програм існують пакети розширення з можливостями:

cc-mode – підсвічування синтаксису C, C++, Objective C, навігація по коду за допомогою CTags;

build – функції інтегрованої компіляції;

debug – інтегроване налагоджування програм за допомогою GDB (GNU Debugger, мови C, C++, Pascal та інші), DDD (Data Display Debugger, графічний інтерфейс до GDB), pydb (Python Debugger), JDB (Java Debugger);

prog-modes – підтримка мов Tcl, Pascal, AWK, JavaScript, Eifell, PHP, SQL; інтеграція з системами CVS, autoconf та програмою diff;

perl-modes, python-modes, ruby-modes, fortran-modes, haskell-mode – модулі підтримки відповідних мов програмування.

Коротка довідка про функції розробки програм:

'M-x compile' – виклик компілятора чи команди make (<M-x> = = <Alt-x>);

'M-x dgb' – виклик інтегрованого налагоджувача;

'ESC:' (setq gdb-command-name "mygdb") – встановлення імені програми-налагоджувача (наприклад, при роботі в UNIX можна вказати програму DDD, яка є значно зручнішим засобом налагоджування).

2. ІНТЕРПРЕТОВАНІ МОВИ ПРОГРАМУВАННЯ

2.1. Мова програмування TCL

Мова Tcl – процедурна інтерпретована мова високого рівня: програми на Tcl готові до виконання без компіляції та компоновки, хоча існує компілятор Tcl. Інтерпретатор Tcl є набором функцій мови С, які можна викликати з коду застосування. В результаті Tcl може використовуватись як макромова всередині застосування. Основні характеристики Tcl:

розширювана мова: в неї легко додаються нові команди написані на С, інтерпретатор Tcl надає можливість створення та автоматичного завантаження бібліотек користувача;

інтегрована мова: команди або програми Tcl можна викликати з програм з іншими мовами програмування;

мобільна мова: Tcl працює на різних платформах.

Файл з текстом програми може бути оформлений як програма UNIX, якщо його перший рядок містить:

```
#!/usr/bin/tcl
```

або

```
#!/usr/bin/env tcl
```

Tcl підтримує тільки один тип даних – рядки. Команди, аргументи команд, результати їх виконання та значення змінних є рядками. Існують три загальні форми рядків: команди, вирази та списки. Знак '#' перед ім'ям команди утворює коментар, усі наступні символи до початку нового рядка інтерпретатором ігноруються.

Підтримка UNICODE. Починаючи з версії 8.1, усі рядки в Tcl оброблюються в кодуванні 'UTF-8'. При здійсненні комунікацій з операційною системою (робота з файлами, введення та виведення даних) Tcl автоматично перекодує рядки в локальне кодування системи або з нього.

Рядок може бути заданий як послідовність кодів 'Unicode' (ISO10646):

```
tcl>set t "\u041f\u0440\u0438\u0432"
tcl>puts $t
   Прив
```

Для конвертації рядків передбачаються функції encoding convertto і encoding convertfrom:

```
tcl>encoding convertfrom cp1251 "Ïšéâåó"
   Привет

tcl>encoding convertto cp1251 "Привет"
   Ïšéâåó
```

Синтаксис команд. Програма Tcl складається з однієї або декількох команд, розділених проміжком (пробіл, табуляція). Перше поле повинно бути ім'ям команди; необов'язкові наступні поля є аргументами, які передаються команді. Наприклад, команда `set a 22` має три поля: перше (`set`) – ім'я команди, останні (`a 22`) – аргументи. Ім'я команди повинно бути іменем вбудованої команди Tcl, додаткової команди, створеної для даного застосування процедурою 'Tcl_CreateCommand' або командною процедурою, визначеною командою 'proc'.

Аргументи передаються текстовими рядками. Команди користуються ними так, як їм потрібно. Наприклад, 'set' розглядає перший аргумент як ім'я змінної, а другий – рядковим значенням для присвоювання цій змінній. Інші команди можуть розглядати аргументи як цілі числа, імена файлів або команди Tcl.

Групування і підстановка аргументів. Групування аргументів виконується з метою отримання спеціального ефекту від застосування аргументів і завдається використанням символів 'подвоєні лапки', 'фігурні дужки', 'квадратні дужки'. Підстановка виконується для заміни імені змінної її значенням.

Подвоєні лапки використовують для формування аргументів з пробілами всередині. Якщо поле аргумента починається зі знаку ", то воно завершується тільки наступним символом ". Якщо лапки не є першим символом у полі команди, то вони не мають спеціального значення. Виконання підстановок команд, змінних та підстановок зі зворотним слешем всередині подвоєних лапок буде збережено.

Фігурні дужки, на відміну від 'подвоєних лапок', допускають вкладення, тому їх можна використовувати для вкладених команд Tcl; підстановки для команд, змінних та зворотних слешей всере-

дині фігурних дужок не виконуються. Якщо поле аргумента починається з відкриваючої фігурної дужки, то воно завершується закриваючою дужкою. Інформація всередині блоку '{}' передається без модифікації (включаючи символи нового рядка). Якщо перший символ поля команди не '{' або '}', то дужки не мають спеціального значення.

Квадратні дужки використовуються для підстановки результату виконання команди в цих дужках замість аргументу, що її утримує. Якщо в полі команди зустрічається відкриваюча квадратна дужка, то всі символи всередині дужок приймаються як команда, що одразу виконується.

Команда в квадратних дужках може містити в собі декілька команд, розділених між собою звичайним способом – символами нового рядка або ';'.

Символ '$' можна використовувати в якості скороченої форми підстановки змінних. Послідовність символів від '$' до першого символа, що не є цифрою, буквою або підкреслюванням, розглядаються як ім'я змінної, і її значення підставляється замість імені. Наприклад, якщо змінна 'foo' має значення 'test', то команда `set a $foo.c` є еквівалентом команди `set a test.c`

Якщо наступним символом за ім'ям змінної є відкриваюча кругла дужка, то змінна розглядається як ім'я масиву, а всі символи до наступної закриваючої дужки – як індекс елемента масиву. Команди та змінні, які використовуються в якості індексу, обробляються перед операцією здобуття елемента масиву. Наприклад, якщо змінна 'x' є масивом та один його елемент з ім'ям 'first' має значення '**87**', то команда `set a xyz$x(first)zyx` є еквівалентом команди `set a xyz87zyx`

Якщо після символа '$' стоїть відкриваюча фігурна дужка, ім'я змінної складається з усіх символів всередині '{}'. Наприклад, якщо змінна 'foo' має значення 'test', то команда `set abc${foo}bar` є еквівалентом команди `set abctestbar`

Зворотний слеш можна використовувати для вводу недрукованих символів у поля команд і для вставки спеціальних символів (наприклад, фігурних дужок або подвоєних лапок). Його використання подібне до запису ESC-послідовностей у C.

Вирази. Дозволені в Tcl-виразах оператори складають підмножину операторів, які можна використовувати у виразах C, і вони

мають такий же зміст і пріоритет виконання, як і відповідні їм оператори С. Майже завжди значенням виразу є число (ціле чи дійсне). Наприклад, вираз `8.2 + 6` дає результат '14,2'. Вирази Tcl відрізняються від виразів С записів операндів, підтримкою нечислових операндів з можливістю порівняння рядків.

Вирази Tcl складаються з комбінації операндів, операторів і дужок. Між ними можна ставити пробіли. По можливості всі операнди інтерпретуються як цілі числа, якщо не задане інше. Цілі числа можуть мати вигляд десяткового числа (звичайно), вісімкового (перша цифра числа – '0') чи шістнадцяткового (перші два символи числа – '0x'). Якщо операнд не підпадає під жоден з названих форматів, він вважається дійсним числом, якщо це можливо. Дійсні числа можна задавати способами, аналогічними мові С. Виключення складає заборона в більшості версій використання суфіксів 'f', 'F', 'l' і 'L'. Якщо числова інтерпретація неможлива, то операнд вважається рядковим і працювати з ним може тільки обмежений набір операторів.

Операнди можуть бути задані одним з наступних способів:

числовим значенням – цілим або дійсним;

змінною Tcl в нотації зі знаком $. У якості операнда буде використане значення змінної;

рядком символів у подвійних лапках. Аналізатор виразів виконає підстановки змінних, команд зі зворотним слешем; отримане значення буде використано в якості операнда;

рядком у фігурних дужках. Усі символи між відкриваючою і закриваючою дужками будуть вважатися операндом без будь-яких підстановок;

командою Tcl у кутових дужках. Команда буде виконана, а її результат буде використаний у якості операнда.

Список – це звичайний рядок з подібною списку структурою, що складається з полів, розділених проміжками. Наприклад, рядок 'Hello World' є списком, що має два елементи (поля). Основна структура списків аналогічна структурі командних рядків за винятком того, що символ нового рядка служить таким же роздільником, як і пробіл з табуляцією. Для списків діють такі ж правила стосовно фігурних дужок, подвійних лапок і зворотніх слешей, як і для команд. Для списків ніколи не виконуються підстановки команд і змінних.

Команди Tcl 'concat', 'foreach', 'lappend', 'lindex', 'linsert', 'list', 'llength', 'lrange', 'lreplace', 'lsearch' і 'lsort' дозволяють складати списки, витягати з них елементи, переглядати зміст і виконувати інші щодо списків функції.

Результати команд. Кожна команда повертає два результати: код і рядок. Код служить для індикації успіху виконання команди, а рядок надає додаткову інформацію. Діючі значення кодів визначені у файлі 'tcl.h':

TCL_OK – код нормального завершення, який повертається за успішного виконання команди. Рядок містить значення, що повертається командою;

TCL_ERROR вказує на помилку. Рядок містить повідомлення, що описує помилку. Глобальна змінна 'errorInfo' надасть текстуальну інформацію про команди і процедури, які виконувалися при виникнені помилки, глобальна змінна 'errorCode' матиме машинно-залежні дані про помилку.

Скалярні змінні і масиви. Tcl підтримує змінні двох типів: скаляри і масиви (вектори). Скалярна змінна має тільки одне значення в кожен момент часу, тоді як змінна-масив може містити будь-яку кількість елементів, що мають ім'я ("індекс") і значення. Індексами масиву можуть бути рядки, необов'язково числового виду. Для посилання на індекси в командах Tcl використовуються круглі дужки. Наприклад, команда 'set x(first) 44' змінить значення елемента масиву 'x' з ім'ям 'first' на нове: '44'. Двовимірні масиви можна імітувати використанням індексів, які складаються з декількох складених разом частин. Наприклад, команди

```
set a(2,3) 1
set a(3,6) 2
```

задають елементи масиву 'a' з індексами '2,3' і '3,6'.

Команда 'array' надає набір засобів для роботи з масивами, у тому числі одержання списку всіх елементів масиву і перегляд їх по одному.

Змінні можуть бути локальними або глобальними. Якщо ім'я змінної застосовується тоді, коли не виконується процедура, то воно автоматично відноситься до глобальних змінних. Імена змінних, що використовуються всередині процедури, звичайно посилаються на локальні змінні, асоційовані з процесом виконання даної процедури. Локальні змінні знищуються по закінченні роботи процедури. При виконанні процедури для вказівки на те,

що ім'я є ім'ям глобальної змінної, використовується команда 'global'.

Процедури. Tcl дозволяє розширити командний інтерфейс за рахунок визначення процедур командою 'proc'. Процедуру Tcl можна викликати для виконання так само, як і будь-яку іншу команду Tcl (у неї є ім'я і вона може мати один чи більше аргументів). Єдиною відмінністю є те, що вона не складається з коду С, скомпонованого всередині програми, а є рядком, який містить одну або декілька команд Tcl:

```
proc test { a b } {
    set p [ expr $a * $b ]
    return $p
}

set g [test 3 5]
puts $g
```

Розширення функціональності Tcl (приклад програми). Додатково до вбудованих команд, що надає бібліотека Tcl, можна визначати свої власні команди мовою програмування С (наприклад, команду обчислення гіпотенузи):

```
#include <tcl.h>
#include <math.h>
int GipotCmd(ClientData clientData, Tcl_Interp *interp,
    int objc, Tcl_Obj *CONST objv[])
{
    Tcl_Obj  *resultptr;
    double    x,y,result;
    int       error;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp,2,objv,
            "Usage: gipot x y");
        return TCL_ERROR;
    }
    error = Tcl_GetDoubleFromObj(interp, objv[1], &x);
    if (error != TCL_OK) return error;
    error = Tcl_GetDoubleFromObj(interp, objv[2], &y);
    if (error != TCL_OK) return error;
    result = sqrt(x*x+y*y);
```

```
    resultptr = Tcl_GetObjResult(interp);
    Tcl_SetDoubleObj(resultptr,result);
    return TCL_OK;
}

int Test_Init(Tcl_Interp *interp) {
    Tcl_CreateObjCommand(interp, "gipot", GipotCmd,
        (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
    return TCL_OK;
}
```

Після компіляції програми в подільну бібіліотеку (з файла 'test.c'):

```
gcc test.c -lm --shared -o test.so
```

Команда може бути використана наступним чином:

```
load ./test.so
gipot 3 4
    5.0
gipot 6.7 8
    10.4350371346
gipot 1 2 3
    wrong # args: should be "gipot 1 Usage : gipot x y"
```

2.2. Мова програмування Python

Python – об'єктно-орієнтована інтерпретована мова програмування з динамічним контролем типів. Програма складається з операторів, які розподіляються символом нового рядка або ';'. Блоки програмного коду (складені оператори) формуються відступом. Розмір відступів не регламентований, але повинен бути однаковим.

Інтерпретатор Python може виконувати код, уведений зі стандартного вводу, та код, розміщений у файлах. Файл з текстом програми може бути оформлений як програма UNIX, якщо для файла встановлена ознака на виконання і його перший рядок містить

```
#!/usr/bin/env python
```

або

```
#!/usr/bin/python
```

Python характеризується великою кількістю бібліотек (модулів), у тому числі для роботи з текстом, файлами, системними змінними, мережами, графікою, базами даних, інтернетом тощо. Підключення модуля виконується (для модуля 'sys'):

```
import sys
print.sys.argv
    ['']
```

Синтаксис Python чутливий до регістру і включає в себе ідентифікатори (імена, що використовуються для позначення змінних, функцій, класів, модулів і інших об'єктів); зарезервовані слова (слова, які особливим чином інтерпретуються Python); літерали (значення, що використовуються в програмі); оператори, подільники й спеціальні символи.

Символ '#' починає коментар, який діє до кінця поточного рядка програми.

Зарезервовані слова використовуються інтерпретатором для реалізації можливостей мови програмування і не можуть бути використані як ідентифікатори. Зарезервовані слова Python:

```
and         elif        global      or
assert      else        if          pass
break       except      import      print
class       exec        in          raise
continue    finally     is          return
def         for         lambda      try
del         from        not         while
```

Типи даних Python. До даних Python відносять літерали (числові та рядкові), кортежі, списки і словники. Літерали й кортежі представляють незмінні об'єкти програми, списки і словники – змінні об'єкти.

Існують чотири вбудовані числові типи:

цілі числа:

число '234' інтерпретується як десяткове ціле число;

число '0234' інтерпретується як вісімкове число;

число '0x234' інтерпретується як шістнадцяткового число;

довгі цілі числа:

число '234L' інтерпретується як довге ціле число, на відміну від цілих чисел довгі цілі можуть мати довільну довжину;

числа з плаваючою крапкою:

числа '23.4' і 23.4+e2 інтерпретуються як числа з плаваючою крапкою;

комплексні числа:

число 234J або 234j або 23.4J або 23.4j представляють уявні числа;

числа 2+34j або 2+3.4j представляють комплексні числа.

Рядок – послідовність символьних літералів, вміщених у лапки ('), подвійні (") або потрійні (''' або """). Для позначення початку і закінчення рядка повинні використовуватися лапки одного типу. Рядки можуть включати ESC-послідовності, що оформлюються в стилі мови С. Усі види лапок рівноправні, потрійні лапки продовжують свою дію на кілька рядків програмного коду. Для створення рядкових літералів використовується, наприклад:

```
a = 'Hello'
b = "World"
c = '''Proba\n
    pera'''
```

Кортежі та списки є послідовностями довільних об'єктів. Кортеж визначається оператором '()', список – оператором '[]':

```
names = ('Dave','Ann','None')
color = ['green','white']
```

Кортежі та списки індексуються цілими числами, починаючи з нуля. Як виняток, кортеж з одного елемента визначається наступним чином:

```
one = ('first',)
```

Словник – це асоціативний масив, який містить пари ключ-значення. В якості ключа може виступати будь-який незмінний об'єкт (тобто списки та словники не припустимі):

```
bob = {
    'name' : 'Bob',
    'address' : {'street': 'Super', 'flat': 5}
}
```

Операції над даними. Дані в Python представлені об'єктами. Звернення до методів об'єкту здійснюється за допомогою операції '.'.

Послідовності (рядки, кортежі і списки)

```
s = 'This is a test'
k = ('This','is','a','test')
l = ['This','is','a','test']
```

підтримують такі операції і методи:

операція індексування (індекс повинен бути цілим числом, додатним – індексування виконується з початку, або від'ємним – індексування виконується з кінця):

```
print s[1]
    h
print k[-1]
    test
print l[0]
    This
```

операція виділення зрізу (індекси повинні бути двома цілими числами, які вказуються через двокрапку; один з індексів може бути опущений – приймається '0' для лівого і значення довжини для правого індекса):

```
print s[1:3]
    hi
print k[:3]
    ('This', 'is', 'a')
print l[1:]
    ['is', 'a', 'test']
```

операція складання:

```
k1 = ('one',)
k2 = ('two',)
k3 = k1 + k2
print k3
    ('one', 'two')
```

методи 'len', 'min', 'max' (методи 'min', 'max' повертають перший об'єкт, що задовільняє вимогам пошуку):

```
len(s)
    14
min(k)
    a
max(l)
    this
```

Рядки підтримують методи:

```
`вираз`          #обчислити вираз (зворотні лапки)
repr(вираз)      #те ж саме, що й `вираз`
str(вираз)       #отримує рядок для обчисленого виразу
```

Списки підтримують методи:

```
list(s)          #перетворює послідовність на список
range(i,j[,step]) #створює список послідовних цілих
s.append(x)      #приєднує новий елемент
s.extend(n)      #приєднує новий список
s.count(x)       #підраховує кількість входжень
s.index(x)       #повертає найменший номер
s.insert(i,x)    #вставляє елемент по індексу
s.pop([i])       #знімає елемент (останній елемент)
s.remove(x)      #видаляє елемент
s.reverse()      #переставляє в зворотньому напрямі
s.sort([cmpfunc]) #сортує елементи (використовує функцію
                  порівняння)
```

Словники підтримують методи:

```
len(m)           #повертає число елементів
m[k]             #повертає елемент з ключем 'k'
m[k]=x           #встановлює значення для ключа 'k'
del m[k]         #видаляє елемент
m.clear()        #видаляє всі елементи
m.copy()         #створює копію словника
m.has_key(k)     #повертає 1, якщо є ключ 'k', інакше – 0
m.items()        #повертає список пар 'ключ, значення'
m.keys()         #повертає список ключів
m.update(b)      #додає всі об'єкти з 'b' у 'm'
m.values()       #повертає список всіх значень в 'm'
m.get(k[,f])     #повертає елемент 'm[k]', якщо немає, то – 'f'
```

Підтримка UNICODE. У Python версії 2.0 до найсучасніших з'явилася можливість використовувати рядки символів у Unicode (кодування ISO10646). Рядок у такому кодуванні може бути заданий як послідовність символів або кодів символів (код позначається як \u):

```
u'Hello\u0020World !'
```

Вбудована функція 'unicode()' запроваджує сервіс перекоду-

вання в unicode, а метод 'encode()' дозволяє здійснити зворотнє перекодування рядків у форматі unicode:

```
unicode('Прив', 'utf-8')
    u'\u041f\u0440\u0438\u0432'
u'\u041f\u0440\u0438\u0432'.encode('utf-8')
    '\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2'
print u'\u041f\u0440\u0438\u0432'.encode('utf-8')
        Прив
```

Оператори, змінні та арифметичні вирази. Більшість операторів є подібною операторам мови С. Наприклад:

```
principal = 1000
rate = 0.05
years = 5
year = 1
while year <= years:
    principal = principal * (1 + rate)
    print year, principal
    year = year + 1
```

Оператор 'pass' є пустим оператором. Наприклад:

```
if a < b:
    z = a
elif a == b:
    pass
else:
    z = b
```

Файловий ввід – вивід і формат виводу. Формат виводу може бути встановлений засобами, подібними до функції 'printf' мови С:

```
print "%3d %0.2f" % (year, principal)
```

Файловий ввід – вивід може бути виконаний наступним чином:

```
#створюємо новий об'єкт
f = open('text.txt')
#використовуємо метод об'єкта
line = f.readline()
while line:
    #символ ',' скасовує формування нового рядка
    print line,
```

```
    line = f.readline()
f.close()
#відкриття файлу в режимі запису
f = open('out.txt','w')
f.write('%3d %0.2f' % (year, principal))
```

Функції. Визначення функції складається з ключового слова 'def', імені функції і списку параметрів:

```
def remainder(a, b):
    q = a/b
    r = a - q*b
    return r
```

Параметри функції передаються за значенням і можуть приймати значення за умовчанням:

```
def remainder(a, b=2):
```

Функція може мати рядок документації, який описує її призначення та аргументи. Рядок документації іде за визначенням функції:

```
def remainder(a, b=2):
    '''The remainder of the division.
    a: numerator, b: denominator'''
```

Документація доступна як атрибут '__doc__':

```
print remainder.__doc__
    The remainder of the division
        a: numerator, b: denominator
```

Класи. Визначення класу складається з ключового слова 'class', імені класу, списку класів, які він успадковує (підтримується множинне успадкування), атрибутів і методів класу. Для створення екземплярів використовується функціональна нотація:

```
class MyClass:
    'A simple class'
    i = 100
    __k = 200
    def __init__(self, j):
        self.j = j
    def test(self):
        return self.i + self.j - self.__k
m = MyClass(5)
```

Немає потреби декларувати атрибути (змінні) екземпляра класу (такі, як 'j' у попередньому прикладі). Метод '__init__()' є конструктором класу. Ті змінні, ім'я яких починається з '__' (наприклад, '__k'), є приватними атрибутами класу. Змінна 'self' є показником поточного екземляру класу.

Успадкування декларується як

```
class Descedant(Predeces1, Predeces2, Predeces3):
```

При успадкуванні автоматичний виклик конструкторів базових класів не виконується, за потребою необхідно вказати (для 'Predeces1' з аргументами 'arg1','arg2'):

```
class Descedant(Predeces1, Predeces2, Predeces3):
    def __init__(self,arg1,arg2):
        Predeces1.__init__(self,arg1)
```

Клас без методів може бути використаний як структура (подібно мові С):

```
class Address:
    pass
addr1 = Address()
addr1.street = 'Super'
addr1.flat = '5'
```

Для класів теж визначений атрибут '__doc__':

```
print MyClass.__doc__
    'A simple class'
```

Використання Python з програм С. Приклад програми:

```
#include <python2.2/Python.h>
int main(int argc, char *argv[]){
    FILE *fp;
    //ініціалізація інтерпретатора
    Py_Initialize();
    //виконання коду, заданого рядком
    PyRun_SimpleString("print 2+2\n");
    //виконання коду з файла 'PyRun_SimpleFile())'
    fp = fopen("test.py", "r");
    PyRun_SimpleFile(fp, "test.py");
    fclose(fp);
    //завершення інтерпретатора
```

```
    Py_Finalize();
    return 0;
}
```

Розширення функціональності Python. Приклад модуля:

```
/*  Модуль spam, функція system()
    аналог однойменної функції UNIX
    USAGE:   import spam
         status = spam.mysystem("ls -l")
*/
#include <python2.2/Python.h>
//об'єкт, що виконує обробку виключних ситуацій spam
static PyObject *SpamError;
//реалізація функції system() для Python
static PyObject *spam_my(PyObject *self, PyObject *a)
{
    const char *command;
    int sts;
    //розбір переданих в функцію аргументів
    //рядкове значення заноситься в змінну command
    if (!PyArg_ParseTuple(a, "s", &command))
        return NULL;
    //виконання system()
    sts = system(command);
    //повернення цілого значення інтерпретатору
    return Py_BuildValue("i", sts);
}

//таблиця методів в модулі для інтерпретатора
static PyMethodDef SpamMethods[] = {
    {"mysystem", spam_my, METH_VARARGS,
        "Execute a shell command"},
    {NULL, NULL, 0, NULL}};

//ініціалізація модуля spam
void initspam(void){
    PyObject *m, *d;
    m = Py_InitModule("spam", SpamMethods);
    d = PyModule_GetDict(m);
```

```
    SpamErr = PyErr_NewException("spam.error",
        NULL, NULL);
    PyDict_SetItemString(d, "error", SpamError);
}
```

3. ЗАСОБИ МІЖМАШИННОЇ ВЗАЄМОДІЇ

3.1. Сокети

Операційна система UNIX забезпечує можливість взаємодії процесів, які виконуються на різних комп'ютерах, з'єднаних мережею передачі даних. Ця можливість базується на забезпеченні файлового інтерфейсу для пристроїв (включаючи мережеві адаптери) на основі поняття спеціального файла; механізмах програмних гнізд (сокетів), адресації машини в мережі (IP-адреса), адресації процесу на машині (адрес порта).

Взаємодія процесів на основі сокетів побудована на моделі "клієнт-сервер". Процес-сервер "слухає (listen)" своє програмне гніздо, а процес-клієнт намагається спілкуватись з ним крізь інше програмне гніздо. При цьому процеси можуть виконуватись на одному або різних комп'ютерах. Ядро підтримує внутрішні з'єднання та маршрутизацію даних від клієнта до сервера і навпаки. Гнізда можуть бути:

з віртуальним з'єднанням (stream sockets). Забезпечують передачу даних від клієнта до сервера в вигляді безперервного потока байтів з гарантією доставки. До початку передачі даних повинно бути встановлено з'єднання, яке підтримується до кінця комунікаційної сесії;

датаграмні гнізда (datagram sockets). Забезпечують високу швидкість передачі без гарантії надійної послідовної доставки пакетів даних (датаграм). Для використання датаграмного режиму не потрібно встановлення з'єднання.

Операційна система вибирає придатний протокол передачі даних для кожної допустимої комбінації "домен-гніздо". Для віртуальних з'єднань вибирається протокол TCP, для датаграмних – UDP.

Функції для роботи з програмними гніздами

int socket(int domain, int type, int protocol) створює нове програмне гніздо.

Параметри:

domain – домен гнізда (AF_UNIX – домен UNIX; AF_INET – домен інтернет);

type – тип гнізда (SOCK_STREAM – з віртуальним з'єднанням; SOCK_DGRAM – датаграмне);

protocol – бажаний мережевий протокол (0 – автоматичний вибір за вказаним domain/type).

Результат: дескриптор програмного гнізда або –1 у випадку помилки (отриманий дескриптор аналогічний дескриптору файла і може використовуватись у будь-якій функції, що використовує дескриптор файла).

int close(int sd) закриває вказане параметром sd програмне гніздо.

Результат: 0 – нормальне виконання; –1 – помилка.

int bind(int sd, struct sockaddr *nm, socklen_t len) зв'язує програмне гніздо з іменем.

Параметри:

sd – дескриптор створеного раніше гнізда;

nm – адреса структури, яка містить ім'я (ідентифікатор) гнізда, що відповідає вимогам домену і протоколу гнізда;

len – розмір структури nm в байтах.

Результат: 0 – нормальне виконання; –1 – помилка.

int connect(int sd, struct sockaddr *nm, socklen_t len) зв'язує процес-клієнт з існуючим програмним гніздом (у процесі-сервері). Для нормального виконання функції необхідно співпадання домену і протоколу гнізда з дескриптором 'sd' і з іменем 'nm'. Якщо тип гнізда з дескриптором 'sd' є датаграмним, то функція лише інформує систему про адресу призначення пакетів, установлення з'єднання не виконується.

Параметри: аналогічні функції 'bind' з тією різницею, що ім'я програмного гнізда відповідає гнізду на іншій стороні комунікаційного каналу (стороні сервера).

Результат: 0 – нормальне виконання; –1 – помилка.

int listen(int sd, int qlength) інформує систему про те, що процес-сервер планує встановлення віртуальних з'єднань через вказане гніздо.

Параметри:

sd – дескриптор створеного раніше гнізда;

qlength – максимальний розмір черги запросів на встановлен-

ня з'єднання, які повинні бути буферизовані системою, доки їх не вибере процес-сервер.

Результат: 0 – нормальне виконання; –1 – помилка.

int accept(int sd, struct sockaddr *nm, socklen_t *len) вибирає черговий запрос на встановлення з'єднання з вказаним програмним гніздом з боку сервера. Якщо до моменту виконання функції черга запросів на встановлення з'єднання порожня, то процес-сервер відкладається до моменту запроса. Виконання функції призводить до встановлення віртуального з'єднання.

Параметри:

sd – дескриптор гнізда, для якого була виконана функція listen;

nm – адреса структури з ім'ям гнізда клієнта, для якого встановлено з'єднання;

len – розмір структури за адресою nm в байтах до виконання;

Результат: новий дескриптор гнізда, який повинен використовуватись при роботі в рамках даного з'єднання; –1 – помилка.

int shutdown(int sd, int mode) негайно припиняє комунікації або зі сторони приймаючого процесу або з обох сторін. Відрізняється від close тим, що виконання останньої призупиняється до кінця спроб системи доставити вже відправлені повідомлення. Також функція shutdown розриває з'єднання, але не ліквідує дескриптори раніше з'єднаних гнізд (для ліквідації необхідно викликати close).

Параметри:

sd – дескриптор гнізда;

mode – SHUT_RD – приймання стає неможливим; SHUT_WR – передавання стає неможливим; SHUT_RDWR – унеможливлює як приймання, так і передавання.

Результат: 0 – нормальне виконання; –1 – помилка.

int send(int sd, const void *msg, size_t len, int flags) передає повідомлення до іншого програмного гнізда.

Параметри:

sd – дескриптор гнізда зі встановленим з'єднанням;

msg – адреса буфера з даними, які потрібно передати;

len – розмір буфера на який вказує msg;

flags (може бути MSG_OOB) – позачергова передача (повідомлення передаються перед непрочитаними повідомленями, клієнт може отримати спеціальний сигнал і одразу прочитати позачергові дані).

Результат: число реально відісланих байтів (за нормальних умов співпадає зі значенням параметра len).

int recv(int sd, void *buf, size_t len, int flags) отримує повідомлення з програмного гнізда.

Параметри:

sd – дескриптор гнізда зі встановленим з'єднанням;

msg – адреса буфера для розміщення даних;

len – максимальний розмір буфера, на який вказує msg;

flags (може бути MSG_PEEK) переписує повідомлення в буфер користувача без знищення його у системних буферах.

Результат: число реально прийнятих байтів.

Алгоритм взаємодії гнізд з віртуальним з'єднанням. Для передачі та приймання даних через програмні гнізда зі встановленим віртуальним з'єднанням використовується послідовність дій, що приведена на рис. 2.1.

Замість функцій 'send' і 'recv' можна використовувати звичайні файлові системні виклики 'read' і 'write'. Для програмних гнізд вони виконуються аналогічно 'send' і 'recv'. Це дозволяє створювати програми, які не залежать від того, працюють вони зі звичайними файлами, програмними каналами або гніздами.

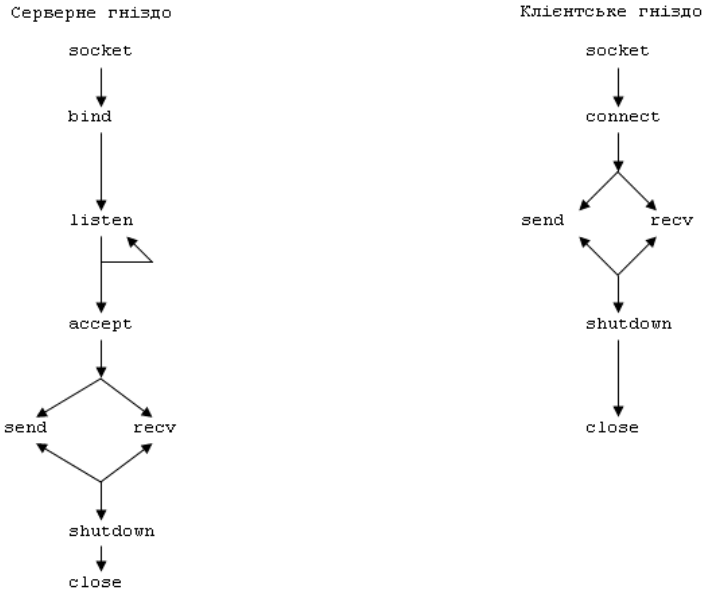


Рис. 2.1. Послідовність дій при встановленні віртуального з'єднання

Алгоритм взаємодії гнізд з датаграмним з'єднанням. У датаграмному режимі використовуються послідовність дій, приведена на рис. 2.2, і функції:

int sendto(int sd, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen) передає повідомлення до іншого програмного гнізда.

Параметри:

sd, msg, len, flags – аналогічно функції send;

to – ім'я програмного гнізда, в яке передається повідомлення (може бути випущено, якщо до цього була викликана функція connect);

tolen – розмір буфера, на який вказує to.

Результат: число реально відісланих байтів (за нормальних умов співпадає зі значенням параметра len).

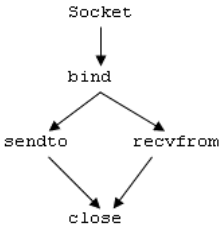


Рис. 2.2. Послідовність дій при встановленні віртуального з'єднання

int recvfrom(int sd, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen) отримує повідомлення з програмного гнізда.

Параметри:

sd, buf, len, flags – аналогічно функції send;

from – ім'я програмного гнізда, з якого були передані дані;

fromlen – розмір буфера, на який вказує from.

Результат: число реально прийнятих байтів.

Приклади програм з використанням програмних гнізд на С

Код програми-клієнта

```
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SIZE sizeof(struct sockaddr_in)
```

```
int main()
{
     int sockfd;
     char c[50], rc[50];
     struct sockaddr_in server = {AF_INET, 7000};
     server.sin_addr.s_addr = inet_addr("127.0.0.1");
     sock = socket(AF_INET, SOCK_STREAM, 0);
     if(sockfd  == -1)
     {
          perror("unable to create socket");
          exit(1);
     }
     if(connect(sockfd,(struct sockaddr *)
          &server,SIZE) == -1)
     {
          perror("unable to connect");
          exit(1);
     }

     for (rc[0] = '\n';;)
     {
          if (rc[0] == '\n') printf("input data:\n");
          gets(c);
          send(sockfd, c, 10, 0);
          if (recv(sockfd, rc, 20, 0)>0) printf("%s",rc);
          else
          {
               printf("communications ended\n");
               close(sockfd);
               exit(1);
          }
     }
}
```

Код програми-сервера

```
#include <ctype.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>

#define SIZE sizeof(struct sockaddr_in)

void catcher(int sig);
int newsockfd;

int main()
{
     int sockfd;
     char c[20];
     struct sockaddr_in server = {AF_INET, 7000,
               INADDR_ANY};
     static struct sigaction act;

     act.sa_handler = catcher;
     sigfillset(&act.sa_mask);
     sigaction(SIGPIPE, &act, NULL);

     if((sockfd = socket(AF_INET, SOCK_STREAM, 0))== -1)
     {
          perror("unable to create socket");
          exit(1);
     }

     if(bind(sockfd, (struct sockaddr *)&server, SIZE)== -1)
     {
          perror("unsuccessful bind");
          exit(1);
     }

     if(listen(sockfd, 5) == -1)
     {
          perror("unsuccessful listen");
          exit(1);
     }
```

```
    for(;;)
    {
        if((newsockfd=accept(sockfd, NULL, NULL))==-1)
        {
            perror("could not accept");
            continue;
        }

        if(fork() == 0)
        {
            while(recv(newsockfd, c, 10, 0) > 0)
            {
                c[0] = toupper(c[0]);
                send(newsockfd, "server
                    response:", 17, 0);
                send(newsockfd, c, 10, 0);
            }
            close(newsockfd);
            exit(0);
        }
        close(newsockfd);
    }
}

void catcher(int sig)
{
    close(newsockfd);
    exit(0);
}
```

Приклади програм з використанням програмних гнізд на Tcl

Програма-сервер

```
#!/usr/bin/env tcl
proc doService {sock msg} {
    puts $sock "$msg"
}
proc svcHandler {sock} {
    set l [gets $sock]
```

```
    if {[eof $sock]} {
        close $sock
    } else {
        doService $sock $l
    }
}
proc accept {sock addr port} {
   fileevent $sock readable [list svcHandler $sock]
   puts "Connection from [fconfigure
            $sock -peername]"

   fconfigure $sock -buffering line -blocking 0

   puts $sock "[exec date]"
}
socket -server accept 8888
vwait events
```

Програма-клієнт

```
#!/usr/bin/env tcl
proc read_sock {sock} {
    global eventLoop
    set date [gets $sock]
    puts stdout "The date is $date"
    close $sock
    set eventLoop "done"
}
set sock [socket "localhost" 8888]
fileevent $sock readable [list read_sock $sock]
fconfigure $sock -buffering line
vwait eventLoop
```

Приклади програм з використанням програмних гнізд на Python

Програма-сервер

```
#!/usr/bin/python
from socket import *
import time
s = socket(AF_INET, SOCK_STREAM)
```

```
s.bind(("",8888))
s.listen(5)
while 1:
    client,addr = s.accept()
    print "Connection from ",addr
    client.send(time.ctime(time.time()))
client.close
```

Програма-клієнт

```
#!/usr/bin/python
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.connect(("localhost",8888))
tm = s.recv(1024)
s.close()
print "The time is ",tm
```

3.2. Remote Procedure Call

***Remote Procedure Call* (*RPC*)** запроваджує засоби виклику віддалених процедур аналогічно виклику локальних. Пакет RPC реалізований як набір бібліотечних функцій (заголовочні файли 'rpc/rpc.h', 'rpc/types.h', 'rpc/xdr.h' та інші).

Для забезпечення незалежності даних від їх внутрішнього машинного зображення використовується окремо специфікований протокол XDR (eXetrnal Data Representation), що визначає стандартний спосіб представлення даних і приховує такі машинно-залежні властивості: порядок байтів в слові, вимоги до вирівнювання початкової адреси структур, формат стандартних типів даних.

Технологія RPC забезпечує можливість виклику віддаленого сервісу як локальної функції (рис. 2.3) та потребує опису інтерфейсів між клієнтом та сервером (типи та розміри даних, їх організація).

Для полегшення розробки програм з використанням RPC пропонується використовувати утиліту 'rpcgen', яка на основі опису інтерфейсу формує клієнтські та серверні інтерфейсні файли та

може формувати шаблони клієнтської та серверної програми. Опис інтерфейсу розміщується в '.x' файлі.

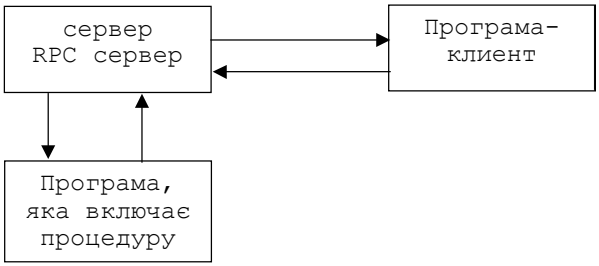


Рис. 2.3. Взаємодія процесів за технологією RPC

Створення найпростішого інтерфейсу

Інтерфейс програми (rpchello.x)

```
program RPCHELLO
{
     version RPCHELLOVERSION
     {
          string HELLO() = 1;
     } = 2;
} = 2000001;
```

Версія програми повинна визначатися з діапазону 2000000…3000000 для запобігання конфліктів з існуючими програмами. При визначенні типів можна використовувати конструкцію typedef.

Формування вихідних файлів програми:

```
rpcgen -a rpchello.x
```

Будуть створені файли:

Makefile.rpchello
rpchello.h – не редагувати
rpchello_svc.c – не редагувати
rpchello_clnt.c – не редагувати
rpchello_server.c
rpchello_client.c

Внесення змін до програми-сервера 'rpchello_server.c' (зміни виділені)

```
#include "rpchello.h"
char **hello_2_svc(void *argp, struct svc_req *rqstp)
{
     static char * result;
     /*
      * insert server code here
      */
     result="Hello, World";
     return &result;
}
```

Внесення змін до програми-клієнта 'rpchello_client.c' (зміни виділені)

```
#include "rpchello.h"
void rpchello_2(char *host)
{
     CLIENT *clnt;
     char * *result_1;
     char *hello_2_arg;
#ifndef DEBUG
     clnt = clnt_create (host, RPCHELLO,
              RPCHELLOVERSION, "udp");
     if (clnt == NULL) {
          clnt_pcreateerror (host);
          exit (1);
     }
#endif   /* DEBUG */
     result_1 = hello_2((void*)&hello_2_arg, clnt);
     if (result_1 == (char **) NULL) {
          clnt_perror (clnt, "call failed");
     }
#ifndef DEBUG
     else printf("%s \n",*result_1);
     clnt_destroy (clnt);
#endif    /* DEBUG */
}
```

```
int main (int argc, char *argv[])
{
     char *host;
     if (argc < 2) {
          printf ("usage: %s server_host\n", argv[0]);
          exit (1);
     }
     host = argv[1];
     rpchello_2 (host);
     exit (0);
}
```

Для виконання програми на стороні сервера повинен існувати демон RPC. Перевірити його присутність можна командою

```
service portmap status
```

запустити –

```
service portmap start
```

зупинити –

```
service portmap stop
```

Для ***опису складних даних*** використовуються об'єднання та структури, наприклад:

```
union resp switch (int rc)
{
     case 0: string Data<>;
     default: void;
};

program RPCUNION
{
     version RPCUNIONVERSION
     {
          resp TESTUNION() = 1;
     } = 2;
} = 2000002;
```

ЛІТЕРАТУРА

1. *Джеймс Армстронг* (*мл.*). Секреты UNIX: 2-е изд.: Пер. с англ.: Учебное пособие – М.: Издательский дом "Вильямс", 2000. – 1072 с.

2. *Билл Болл, Дэвид Питтс.* Red Hat Linux 7.x. Энциклопедия пользователя: Пер. с англ. – С.-Пб.: ООО "ДиаСофтЮП", 2002. – 880 с.

3. *Иван ван Лейнингем*. Освой самостоятельно Python за 24 часа: Пер. с англ. – М. : Издательский дом "Вильямс", 2001. – 448 с.

4. *Джерри Пик, Тим О'Райли, Майк Лукидис.* UNIX: Инструментальные средства: Пер. с англ. – К.: Издательская группа BHV, 1999. – 944 с.

5. *Кейт Хевиленд, Дайна Грей, Бен Салама.* Системное программирование в UNIX. Руководство программиста по разработке ПО = UNIX System Programming. A programmer's guide to software development: Пер. с англ. – М.:, ДМК Пресс, 2000. – 368 с. (сер. "Для программистов").

6. *Теренс Чан.* Системное программирование на C++ для UNIX: Пер. с англ. – К.: Издательская группа BHV, 1997. – 592 с.

7. *Шон Уолтон.* Создание сетевых приложений в среде Linux: Пер. с англ. – М.: Издательский дом "Вильямс", 2001. – 464 с.

ЗМІСТ

Анатолій Юліанович ГАЙДА
Олександр Борисович ДИМО

Інструментальні засоби операційних систем UNIX

Частина I

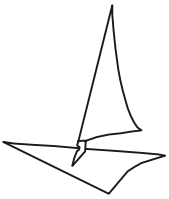
Видавництво УДМТУ, 54002, м. Миколаїв, вул. Скороходова, 5

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції ДК № 1150 від 12.12.2002 р.

Редактор О.Г. Тулузакова
Комп'ютерна правка та верстка Ю.В. Зайцева
Коректор Н.О. Шайкіна

Підписано до друку 04.07.03. Формат 60×84/16. Папір офсетний.
Ум. друк. арк. 2,4. Обл.-вид. арк. 2,6. Тираж 100 прим.
Вид. № 7. Зам. № 270. Ціна договірна.

ВИДАВНИЦТВО УКРАЇНСЬКОГО
ДЕРЖАВНОГО МОРСЬКОГО
ТЕХНІЧНОГО УНІВЕРСИТЕТУ



Шановні панове!

Запрошуємо Вас ознайомитись з можливостями книжкового видавництва, висококваліфіковані спеціалісти якого забезпечать оперативне та якісне виконання замовлення будь-якого рівня складності.

Наш головний принцип – задовольнити потреби замовника в повному комплексі поліграфічних послуг, починаючи з розробки та підготовки оригіналу-макета, що виконується на базі IBM PC, і закінчуючи друком на офсетних машинах.

Крім цього, ми маємо повний комплекс післядрукарського обладнання, що дає можливість виконувати:

- ✓ аркушепідбір;
- ✓ брошурування на скобу, клей;
- ✓ порізку на гільйотинах;
- ✓ ламінування.

Видавництво також оснащено сучасним цифровим дублікатором фірми "Duplo" формату А3, що дає можливість тиражувати зі швидкістю до 130 копій за хвилину.

Для постійних клієнтів – гнучка система знижок.

Отже, якщо вам потрібно надрукувати ***підручники, книги, брошури, журнали, каталоги, рекламні листівки, прайс-листи, бланки, візитні картки,*** *– ми до Ваших послуг.*

© Український державний морський технічний університет
✉ Україна, 54002, м. Миколаїв, вул. Скороходова, 5,
видавництво УДМТУ
☎ 8(0512) 37-33-42; 39-81-46, 39-73-39, fax 8(0512) 39-73-26;
E-mail: publishing@usmtu.edu.ua

ДЛЯ НОТАТОК