УДК 004.75, 004.724.2

Г.В. ПОРЄВ

ДОСЛІДЖЕННЯ МЕТОДІВ РОЗРОБКИ КРОСПЛАТФОРМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Національний технічний університет України «Київський політехнічний інститут», 37, пр.Перемоги, м.Київ, 3703056, Україна
Тел.: (068) 321-345-1, E-mail: core@barvinok.net

Анотація. В статті розглянуті типові проблеми які можуть виникнути при розробці кросплатформного програмного забезпечення. Надано рекомендації щодо їх уникнення.
Аннотация. В статье рассмотрены типовые проблемы, которые могут возникнуть при разработке кроссплатформного программного обеспечения. Предложены рекомендации по их избежанию.
Abstract. In this article the typical problems that may arise when developing the cross-platform applications have been reviewed. The recommendations to amend or avoid such problems have been given.
Ключові слова: кросплатформний, програмне забезпечення.

ВСТУП

Для терміну «обчислювальна платформа» в контексті розробки програмного забезпечення (ПЗ) існують розширені визначення, але в рамках даної роботи ми використаємо спрощене визначення — сукупність апаратного забезпечення та операційної системи (ОС).

Двохкомпонентність обчислювальної платформи є принциповим моментом, оскільки тотожність або відмінність платформи визначається кожним компонентом незалежно. Апаратне забезпечення визначається центральним процесором (ЦП) та його належністю до певного сімейства, що, в свою чергу, визначається сумісністю ЦП з множиною машинних команд. Характеристикою ОС, яка дозволяє визначати тотожність або відмінність платформ, є сумісність бінарного програмного інтерфейсу, тобто, здатності запускати на виконання програмні модулі без використання спеціальних емуляторів.

Проілюструємо вищесказане прикладами. Так, ЦП сімейства Intel Core або AMD Athlon при наявності тієї самої ОС Windows утворюють тотожну платформу, оскільки вони зберігають сумісність з базовою системою команд процесору i386. В той же час ЦП сімейства Intel Itanium з тією самою ОС Windows утворюють відмінну платформу, оскільки цей ЦП несумісний з множиною команд, утворених на базі i386. Один і той же процесор AMD Phenom під керуванням ОС Windows та ОС GNU/Linux утворює відмінну платформу, оскільки кожна з цих ОС не має штатних засобів виконання програмних модулів, створених для іншої ОС. А коли такі засоби існують (наприклад, підсистема POSIX [1] в складі ОС Windows або проект WINE), вони є емуляторами і на сьогоднішній день не забезпечують повної сумісності.

Визначивши обчислювальну платформу, можна перейти до визначення «кросплатформності» — спеціальної властивості вихідного програмного коду, яка дає можливість генерувати програмні модулі (виконуваний машинний код) для відмінних обчислювальних платформ при відсутності або мінімальних автоматизованих змінах до вихідного програмного коду.

Потрібно чітко відрізняти поняття кросплатформності як властивості вихідного програмного коду і поняття рівня забезпечення кросплатформності — властивості компілятора мови програмування, хоча в популярній літературі ці поняття позначаються одним і тим же терміном «кросплатформність», подекуди відносячись до мови, а не до її компілятора.

Прийнято вважати, що мінімально необхідний рівень забезпечення кросплатформності для певної мови програмування досягається тоді, коли компілятор для цієї мови має змогу генерувати програмні модулі в нативному для цільової платформи форматі. Відповідно до двохкомпонентної природи платформи, така нативність повинна включати, по-перше, здатність генерувати машинний код для цільового ЦП, а, по-друге, відповідність структури виконуваних модулів або бібліотек внутрішнім стандартам цільової ОС, тобто відповідність так званому ABI (Application Binary Interface). Наприклад, кросплатформний компілятор може вважатися таким, що підтримує платформу Windows x86 якщо машинний код згенерованого модуля відповідає, як мінімум, множині команд i386, а структура згенерованого модуля відповідає ABI для PE-

формату (Portable Executable), який є стандартним в середовищі ОС Windows.

Прикладами сучасних кросплатформних компіляторів можуть бути GNU Compiler Collection та FreePascal Compiler.

В стандартах одних з найбільш поширених у недалекому минулому мов програмування С та С++ закладено рівень абстракції від обчислювальної платформи у вигляді так званої стандартної бібліотеки. Заголовочні файли для стандартної бібліотеки повинні бути присутні в будь-якому середовищі розробки для цих мов [2], а сама стандартна бібліотека повинна бути присутня в середовищі виконання або як штатна компонента, або як супровідна бібліотека в складі ПЗ.

При умові, що компілятори притримуються одного і того ж діалекту мови програмування а також при умові, що програмний код не використовує виклики функцій ОС або інших бібліотек безпосередньо, це, теоретично, дозволяє проводити компіляцію вихідного коду під різні цільові обчислювальні платформи без змін. На практиці, однак, задачі, які можуть бути повністю вирішені програмним кодом в такому стилі, мають суто академічний характер і трапляються дуже рідко. Це може бути зумовлено необхідністю надання користувацького інтерфейсу або встановлення системних служб, відмінністю роботи файлових систем, навіть відмінністю реалізації стандартних бібліотек від різних вендорів тощо. З огляду на практичні міркування функціональність стандартної бібліотеки обмежена задачами високого ступеню абстракції, наприклад — керування пам'яттю, робота з файлами, масивами та векторами тощо.

Як правило, середовище виконання надає можливість користуватися програмними інтерфейсами операційної системи (або віртуальної машини) а також завантажувати додаткові програмні бібліотеки. Таким чином реалізована можливість користування додатковими бібліотеками, в тому числі і тими, які надаються сторонніми вендорами. Додаткові бібліотеки зазвичай мають вузьку спеціалізацію порівняно зі стандартною, наприклад — набір криптографічних функцій, спеціальна обробка звуку, завантаження файлів з мережі, парсинг XML-файлів тощо.

Для різних операційних систем можуть існувати як дистрибутиви одних і тих же додаткових бібліотек, так і різні бібліотеки зі схожою функціональністю. З точки зору ефективності виконуваного коду суттєва різниця між цими двома підходами практично відсутня. Розглянемо детальніше питання створення кросплаторменого коду і проблеми, які можуть виникнути при використанні декількох аналогічних інтерфейсів чи бібліотек.

РОЗРОБКА МЕТОДУ УЗАГАЛЬНЮЮЧОГО КЛАСУ ДЛЯ БІБЛІОТЕК CURL ТА WININET

Візьмемо прикладну задачу універсального завантажувача файлів з мережі через протоколи HTTP або FTP. Від виклику функції, що ініціює завантаження до надходження відповідних пакетів з мережного пристрою запит проходить кілька рівнів. З урахуванням відмінностей між ОС Windows та GNU/Linux можна умовно виділити такі рівні: інтерфейс додаткової бібліотеки завантаження — внутрішня обробка з використанням сокетів — стек TCP/IP операційної системи — драйвер мережного пристрою. Таке виділення не відповідає стандартній моделі OSI [3] але відображає суть того, що відбувається в системі.

В операційних системах, які історично походять з сімейства ОС UNIX або мають аналогічну архітектуру, яка відповідає стандарту POSIX, для даної задачі традиційно використовують бібліотеку CURL, яка входить в усі сучасні дистрибутиви ОС GNU/Linux для персональних та серверних систем. Ця бібліотека самостійно виконує обробку переданого URL, викликає функції сокетів, реалізує протоколи передачі, контроль цілісності, обробку помилок тощо.

Також існує версія бібліотеки CURL для ОС Windows та багатьої інших операційних систем. На перший погляд, розробнику програмного забезпечення достатньо створити програмний код, що використовує бібліотеку CURL, і позначити наявність цієї бібліотеки як системну вимогу (або, у випадку ОС Windows, постачати продукт з бібліотекою CURL у вигляді DLL). Таким чином можна забезпечити одноманітність вихідного коду у всьому проекті.

Попри всі очевидні переваги такого підходу, у нього є і суттєві недоліки. По-перше, за рахунок включення додаткової бібліотеки збільшується розмір дистрибутиву ПЗ для ОС Windows. По-друге, бібліотека CURL не має можливості самостійно дізнатися про специфіку підключення конкретного комп'ютера до мережі Інтернет, наприклад параметри та умови використання проксі-серверів, які можуть залежати від запитаної адреси, від часу запиту, обчислюватися за допомогою протоколу WPAD (Web-Proxy Auto-Discovery), або потребувати інтерактивної авторизації користувача за допомогою NTLM. По-третє, дистрибутиви CURL реалізовані не для всіх навіть популярних цільових платформ. Наприклад, підтримка ОС Windows Mobile наразі відсутня і малоімовірно, що буде реалізована найближчим часом. Аналогічні обмеження існують також і для інших бібліотек, які мають відповідні аналоги в інтерфейсі операційних систем.

Але в ОС сімейства Windows, в тому числі Windows Mobile, є аналогічна CURL за функціональністю бібліотека WinInet, реалізована як підмножина Windows API. WinInet використовується як в прикладному так і в серверному ПЗ як універсальний (в межах сімейства Windows) засіб отримання файлів з мережі через протоколи HTTP та FTP, а також деталізованої композиції та надсилання HTTP-запитів. Оскільки принципи роботи CURL та WinInet здебільшого однакові, було поставлено задачу створити вихідний код допоміжного класу, який в залежності від цільової ОС при компіляції використає ту чи іншу бібліотеку. Використання функцій обох бібліотек в загальному випадку вкладається в наступну схему: ініціалізація — конфігурація — відкриття ресурсу — надсилання запиту — очікування відповіді — зчитування відповіді — фіналізація. Зауважимо, що як CURL так і WinInet не потребують викликів для усіх і кожного з перерахованих етапів, але множина потрібних викликів у них, природньо, не співпадає. Крім того, бібліотека CURL надає дві окремі множини API — «легку» і «важку», які відрізняються можливістю асинхронного спостереження за процесом запиту за зчитування відповіді тощо. Отже, при проектуванні зовнішнього інтерфейсу допоміжного класу потрібно врахувати, які виклики функцій у обох бібліотек мають спільне семантичне значення в схемі роботи та об'єднати їх у програмному коді зі збереженням простоти, наочності та інтуїтивної зрозумілості методів та властивостей класу.

Для вирішення задачі знаходження локальності нам потрібно створити такий допоміжний клас, який би автоматизував завантаження файлів з базами даних по даному URL і враховував вищенаведений принцип кросплатформності. Нами було створено такий клас, який має 2 властивості (адресу URL та процедуру обробки наступного буфера даних) і 1 метод для ініціації завантаження. Оголошення інтерфейсу у вихідному коді цього класу наведена на рисунку 1.

```
TNetDownloader = class
 constructor Create(const UserAgent : WideString);
 destructor Destroy; override;
private
 // shared
 FURL : AnsiString;
 FBuf : Pointer;
 FBufLen : Cardinal;
 EHandler : TDownloadHandler;
 // platform-dependent
 {$IFDEF WINDOWS}
 hInet, hURL : THandle;
 {$ENDIF}
 {$IFDEF LINUX}
 CURL : Pointer;
 {$ENDIF}
 procedure SetURL(const NewURL : AnsiString);
public
 function Download : LongBool;

 property URL : AnsiString read FURL write SetURL;
 property Handler : TDownloadHandler read EHandler write EHandler;
end;
```

Рис. 1. Оголошення інтерфейсу

Попри використання «легкої» частини інтерфейсу CURL, відсутності асинхронного спостереження і можливості детальних налаштувань, такий клас виконує свою функцію повністю і, з точки зору програміста і користувача, абсолютно прозоро щодо сімейства ОС, при умові наявності в ній відповідних бібліотек. Таким чином, досягається кросплатформність коду при його незмінності і вирішується одна з допоміжних задач для знаходження локальності.

РОЗРОБКА МЕТОДУ УНІВЕРСАЛЬНОЇ БІБЛІОТЕКИ ДЛЯ MSXML ТА LIBXML2

Проілюструвати авторське розуміння кросплатформності можна на прикладі задачі парсингу XML-файлів, хоча програмні реалізації, які вирішують цю задачу, були створені сторонніми розробниками. Група технологій і протоколів, які зосереджені навколо та базуються на множині стандартів щодо XML на сьогоднішній день дуже сильно розвинена. Цьому сприяє зручність використання XML для задач програмної взаємодії компонентів ПЗ, які не були початково спроектовані для такої взаємодії. Оскільки XML в основі своїй є текстовим форматом розмітки, будь-яке ПЗ, що використовує XML для збереження або зчитування даних, обов'язково включає виклик функції парсера. З огляду на широкий вжиток XML природнім рішенням для виробників операційних систем стало включення бібліотек парсингу XML до складу операційних систем. В ОС сімейства Windows це є бібліотека Microsoft Core XML Services

(MSXML), в інших операційних системах часто використовується бібліотека libxml2.

Парсери по своїй природі не використовують специфічні програмні інтерфейси ОС поза керуванням пам'яттю та читанням вихідного файлу, їх задача полягає лише в перетворенні представлення даних, яке може бути виконане цілком в контексті внутрішніх даних екземплярів класу парсера. Це означає, що при проектуванні кросплатформного ПЗ немає нагальної потреби закладати виклик тієї чи іншої бібліотеки в залежності від цільової ОС, як це зроблено вище, так як відмінність між ними полягає лише в способі використання їх інтерфейсів та, можливо, невеликої різниці в швидкодії завдяки оптимізації вихідного коду.

Перевагою MSXML в даному випадку могло б бути те, що ця бібліотека є обов'язковим системним компонентом Windows, до якого в рамках програми Windows Update періодично з'являються оновлення для відповідності новим стандартам та виправлення помилок, і це здійснюється в автоматичному режимі. В той же час, механізми оновлення бібліотеки libxml2 наявні далеко не в усіх дистрибутивах ОС. Навіть там, де така можливість передбачена, тобто існує централізований репозиторій ПЗ, що постійно оновлюється (наприклад Linux-дистрибутиви Fedora, Gentoo, Debian), відповідні функції не завжди активовані. Якщо ж прийняти libxml2 як незмінну компоненту кросплатформного ПЗ, розробники повинні будуть постачати готовий бінарний дистрибутив libxml2 зокрема у версії для Windows, де принципово відсутні засоби для автоматичного оновлення цієї бібліотеки.

З іншого боку, використання в кросплатформному ПЗ для задач парсингу XML тільки libxml2 дає можливість зменшити об'єм коду та підтримувати його однорідність, так як немає потреби передбачати використання іншої бібліотеки і пов'язаної з цим необхідності проектувати узагальнений клас зі спільною для обох бібліотек функціональністю, як це було зроблено вище з бібліотеками CURL та WinInet.

ВРАХУВАННЯ ВІДХИЛЕНЬ ВІД СТАНДАРТУ ПРИ ПРОГРАМУВАННІ СОКЕТІВ BSD

Сучасний стандарт POSIX [1] містить так звані BSD сокети — інтерфейс прикладної бібліотеки для обміну інформацією в комп'ютерних мережах. Всі сучасні операційні системи включають в себе реалізацію цього стандарту, його підмножин або розширень у вигляді бібліотек, які відповідають даному інтерфейсу. Стандарт BSD сокетів має довгу історію, яка розпочалася в університеті Берклі в 1983 році, і був початково спроектований, як і інші розробки для ОС UNIX того часу, з прицілом на кросплатформність.

Дійсно, бібліотеки, що реалізують BSD сокети навіть із збереженням оригінальних імен функцій присутні в ОС сімейств Windows, Linux, MacOS та, природньо, BSD. Логічно було б очікувати від цих реалізацій сумісності на рівні програмного коду, який користується сокетами. Проте при видимій відповідності стандарту розробники ОС вимушені також враховувати архітектурну специфіку операційних систем, зокрема механізми міжпроцесної взаємодії та різницю у представленні та обробці файлових та мережних вказівників.

Поки програма, що використовує сокети, не переходить межу складності вище демонстраційно-навчального рівня, не використовує функції керування потоками виконання, не використовує неблокуючий режим сокетів, відмінності ніяк не проявляють себе. Але на розробку більш складних прикладних програм впливають принципові архітектурні відмінності ОС. Однією з перших проблем, що зустрічають розробники ПЗ в такому випадку і при умові, що вони намагаються користуватися лише нативними інтерфейсами операційної системи, є виклик функції select() з інтерфейсу BSD сокетів. Поведінка цієї функції суттєво відрізняється між її реалізаціями в ОС Windows та Linux навіть на рівні інтерпретації переданих параметрів та їх очікуваної структури. Зокрема, системний тип FD_SET має структуру індексованого масиву з окремим полем довжини під ОС Windows та бітового поля під ОС Linux.

Відмінності існують також в поведінці системи при завершенні процесу, що створював сокети, але не викликав їх закриття безпосередньо. В ОС Windows всі породжені процесом керовані ресурси автоматично звільняються та видаляються, тоді як в ОС Linux та BSD завершення потрібно окремо викликати, інакше можуть виникнути проблеми з їх повторним використанням навіть тоді, коли закриття не виконане на віддаленому боці з'єднання. Проблеми проявляються, зокрема, в формі неможливості повторного використання того ж самого номера порту TCP у випадку, коли функцію закриття з'єднання не було викликано внаслідок аварійного завершення програми. Хоча ця неможливість має тимчасовий характер і триває лише кілька хвилин, для високонадійних систем під великим цілодобовим навантаженням такі відмови в обслуговуванні клієнтських запитів можуть спричинити серйозні збитки або порушення цілісності даних.

АРХІТЕКТУРНІ ВІДМІННОСТІ ПЛАНУВАЛЬНИКА ЦП

Базовою одиницею виконання коду в ОС Windows є поток (thread), тобто планувальник

процесорного часу в ядрі операційної системи переключає виконання між потоками, керуючись їх встановленими пріоритетами виконання. Потоки породжуються процесами і виконуються в спільному адресному просторі процесу, маючи змогу безпосередньо використовувати одні і ті ж ділянки пам'яті процесу. В такій архітектурі процес включає в себе, зокрема, таблицю власних ресурсів, виділену робочу пам'ять, віддзеркалені з дискового носія образи виконання з машинним кодом, та множину потоків, які виконуються як ділянки цього коду і мають спільний доступ до пам'яті процеса.

В ОС Linux та BSD початково не було поняття окремого потоку, а базовою одиницею планувальника ЦП в цих операційних системах був процес. Коли необхідно обробляти паралельно декілька схожих задач, ПЗ розробляється таким чином, щоб комбінувати внутрішню паралельну обробку даних в тілі самої програми і розмноження (англ. forking) копій процесів цієї програми.

Однак сучасні версії цих операційних систем можуть включати декілька підходів до реалізації моделі класичних потоків. Це, зокрема, реалізація fibers у просторі користувача, або розширення стандарту POSIX Threads [4] чи Native POSIX Thread Library (NPTL).

З точки зору архітектури планувальника, реалізація NPTL відрізняється від потоків Windows необхідністю включення необов'язкової підтримки NPTL в опції ядра Linux при його компіляції та необхідністю використовувати спеціальні ядерні сигнали (футекси) для синхронізації потоків, їх пробудження та призупинення.

ВИСНОВКИ

Процес проектування програмного забезпечення взагалі, а особливо кросплатформного, завжди включає в себе стадію розбиття проекту на програмні компоненти за принципом виконання специфічної функції. Підсумовуючи вищесказане, можна сформулювати перелік рекомендацій щодо вибору конкретних підходів до проектування таких компонент.

Необхідно:

- створювати узагальнюючий клас з методами, спільними для всіх підлеглих реалізацій певної функції у випадках, коли для всіх цільових платформ існують бібліотеки з аналогічними функціями, які є обов'язково присутніми в системі і які враховують і використовують специфічні для цільової платформи умови роботи ПЗ;
- з множини аналогів обирати одну бібліотеку, яка підтримується або присутня на всіх цільових системах у випадках, коли функціональність аналогів не відрізняється суттєво і задача не є системо-специфічною для цільових ОС;
- ретельно досліджувати і, при необхідності, враховувати відмінності в реалізаціях поширеного стандарту, коли ним не регламентовані деталі реалізацій, якими планується скористатися при розробці ПЗ; зокрема це стосується BSD сокетів.
- враховувати відмінності архітектурно близьких рішень на різних платформах, особливо для випадків, коли певний функціонал є недоступним за специфікою цільової платформи; зокрема, це стосується системного планувальника задач в ОС.

Виконання цих рекомендацій дозволить розробникам ПЗ створювати ефективне та економічне з точки зору витрат часу кросплатформне програмне забезпечення з розділеною кодовою базою, що значно сприяє поширенню рішень як користувацького так і промислового секторів галузі ІТ.

СПИСОК ЛІТЕРАТУРИ

1. System Application Program Interface (API) [C Language]. Portable Operating System Interface (POSIX) [Електронний ресурс] : IEEE Standard 1003.1—1990.—IEEE Computer Society, 1990 : http://standards.ieee.org/catalog/olis/arch_posix.html.
2. Бьярн Страуструп. Язык программирования С++.—М.:БИНОМ, СПб.: Невский диалект, 2001. – 1099 с. ISBN 5-7989-0226-2.
3. Information Processing Systems — Open Systems Interconnection (OSI) — Basic Reference Model [Електронний ресурс]: ISO 7498-1:1994.— ISO, Geneva, Switzerland, 1994 : http://www.iso.org/iso/catalogue_detail.htm?csnumber=20269.
4. Bradford Nichols. PThreads Programming / Bradford Nichols, Dick Buttlar, Jacqueline Farrell.—Sebastopol, CA, USA: O'Reilly Media, 1996.—288 p. ISBN: 978-1-56592-115-3, ISBN 10: 1-56592-115-1.

Надійшла до редакції 21.04.2010р.

ПОРЄВ ГЕННАДІЙ ВОЛОДИМИРОВИЧ – к.т.н., докторант НТУУ «Київський Політехнічний Інститут», Київ, Україна.