ЛЕКЦІЯ 10. ВИКОНУВАНІ ФАЙЛИ

- Статичне компонування виконуваних файлів
- Динамічне компонування та динамічні бібліотеки
- Формат виконуваних файлів ELF
- Розробка динамічних бібліотек у Linux
- Формат виконуваних файлів PE
- Розробка динамічних бібліотек у Windows XP

Розглянемо організацію виконуваного коду в операційних системах. Спочатку зупинимося на створенні виконуваних файлів під час компонування і на тому, як цей процес впливає на структуру таких файлів, потім на понятті динамічної бібліотеки і різних способах завантаження коду таких бібліотек.

Код після його завантаження у пам'ять виконують у рамках процесів і потоків. Організація виконуваного коду визначає структуру адресного простору процесу. Виконуваний код зберігають у файлах, для читання яких потрібно використовувати інтерфейс файлової системи, а під час їхнього завантаження – технологію відображуваної пам'яті.

10.1. Загальні принципи компонування

Компонуванням (linking) називають процес створення фізичного або логічного виконуваного файла (модуля) із набору об'єктних файлів і файлів бібліотек для подальшого виконання або під час виконання і вирішення проблеми неоднозначності імен, що виникає при цьому.

У разі створення фізичного виконуваного файла для подальшого виконання компонування називають статичним; у такому файлі міститься все потрібне для виконання програми. У разі створення логічного виконуваного файла під час виконання програми компонування називають динамічним; у цьому випадку образ виконуваного модуля збирають «на ходу».

Виокремимо основні питання, які потрібно вирішувати під час реалізації компонування.

- Як давати імена і звертатися в коді до неіснуючих об'єктів на цей момент?
- Як з'єднувати різні простори імен у несуперечливе ціле?

Компонування є реалізацією системи іменування. Такі системи відображають імена на значення, при цьому імена об'єктів у вихідному коді мають відображатися на адреси, необхідні для їхнього використання процесором.

10.2. Статичне компонування виконуваних файлів

Об'єктні файли

Під час компонування виконуваний файл будують із *об'єктних файлів* (object files), які створює компілятор. Об'єктний файл має заголовок, що містить розмір ділянок коду і даних, а також зсув таблиці символів; *об'єктний код* (інструкції і дані, згенеровані компілятором), який звичайно розділений на поіменовані ділянки (секції) залежно від призначення; *таблицю символів* (symbol table). Приклад створення об'єктних файлів показано на рис. 10.1.

Таблиця символів – це спеціальний розділ об'єктного файла, що містить визначення зовнішніх імен, які задають імена та відносні адреси файлових об'єктів, призначених для

використання в інших файлах; зовнішні посилання (глобальні символи, що використовуються у файлі), які зазвичай містять зсув відповідної інструкції та необхідний символ.

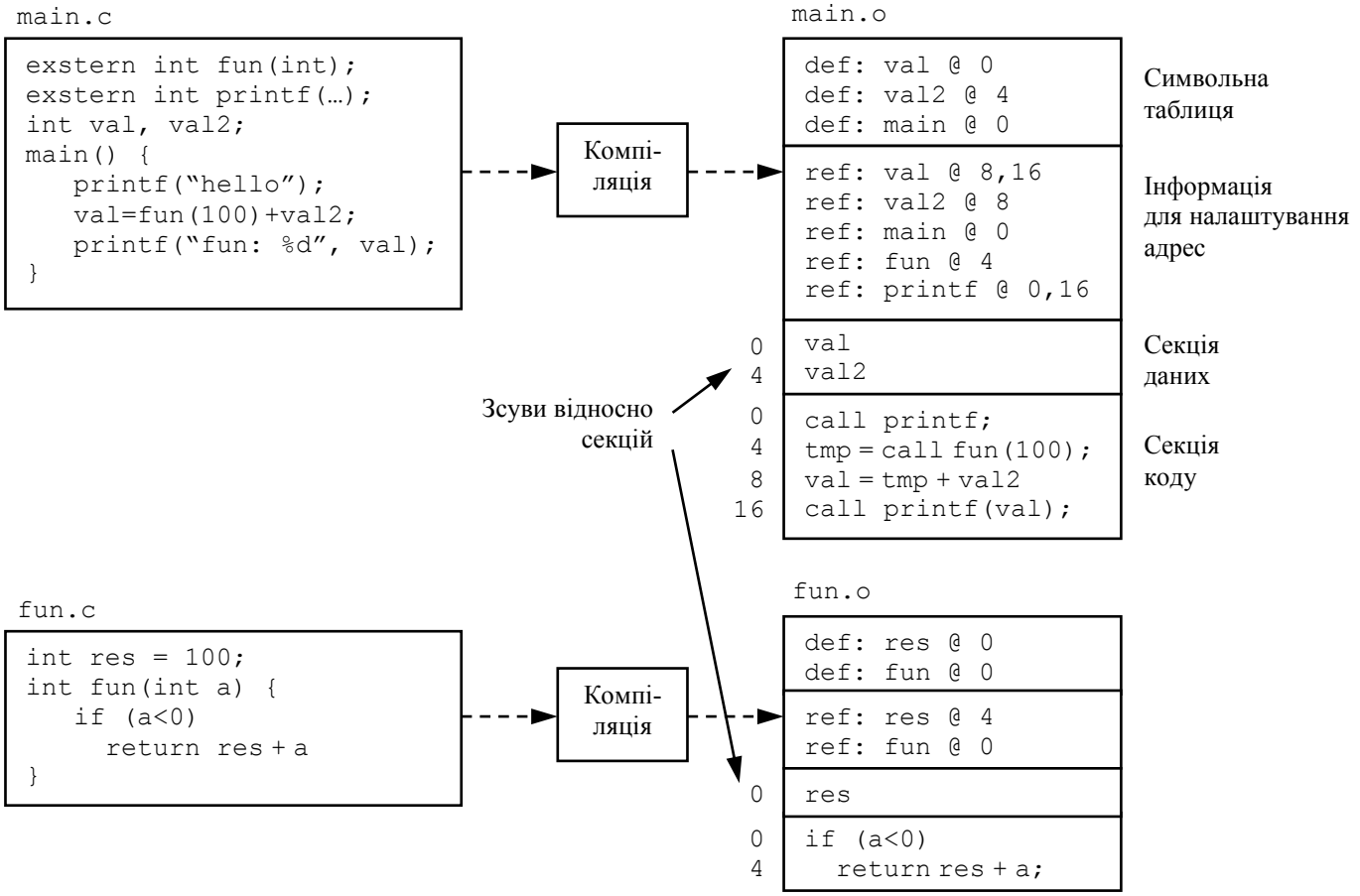

Рис. 10.1. Створення об'єктних файлів

Інформацію про зовнішні посилання називають також інформацією для налаштування адрес (relocation information).

Компілятор створює один об'єктний файл за один запуск, при цьому до інших об'єктних файлів або бібліотек не звертається, тому він ніколи не пов'язує зовнішні посилання із конкретними адресами, тобто не розв'язує їх.

На рис. 10.1, а також на рис. 10.2-10.3 елементи символьної таблиці відображаються так:

- `def: name @offset` – інформація про визначення зовнішнього імені `name`, яке перебуває у файлі зі зсувом `offset`;
- `ref: name @offset` – інформація про зовнішнє посилання на ім'я `name`, що перебуває у файлі зі зсувом `offset`.

Компонувальники і принципи їх роботи

Як зазначалося, компілятор не розв'язує зовнішні посилання, а отже, не може створити виконуваний файл. Це робота *компонувальника* (linker).

Його основні функції такі: об'єднує всі частини програми у виконуваний файл; збирає разом код і дані секцій одного призначення з різних об'єктних файлів; задає адреси для коду і даних, розв'язуючи при цьому зовнішні посилання.

У результаті за статичного компонування на диск записують виконуваний файл, готовий до запуску, за динамічного – виконуваний файл теж буде створено, але йому для виконання потрібні додаткові файли.

Зазвичай компонувальнику для виконання його роботи потрібні два проходи.

1. На першому він збирає разом секції, розподіляє пам'ять, складає глобальну таблицю символів із елементами, що відповідають кожному використовуваному або

визначеному символу. Наприкінці проходу стають відомі адреси всіх секцій виконуваного файла.

2. На другому він коригує кожне посилання в коді з урахуванням інформації про адреси секцій у виконуваному файлі і глобальної таблиці символів, після чого створює виконуваний файл.

Глобальна таблиця символів містить інформацію про програму, що зберігається між проходами компонувальника. За секціями – ім'я, розмір, старе і нове місце розташування; за символами – ім'я, секція, зсув усередині секції.

Зупинимося докладніше на проходах компонувальника.

Головне завдання, яке вирішують на першому проході компонувальника, полягає у визначенні адрес, за якими потрібно розміщувати задані в коді об'єкти. Компілятор під час генерації символьної таблиці не має інформації про те, у якому місці адресного простору процесу потрібно розміщувати дані, а в якому – код; крім того, він вважає, що всі секції починаються від нульової адреси, тому у символьній таблиці зберігаються пари (ім'я, зсув). їх називають глобальними визначеннями. Завданнями компонувальника під час роботи з ними є:

- визначення розміру і місця розташування кожної секції виконуваного файла і розрахунок адреси розміщення кожного об'єкта всередині цих секцій;
- розміщення всіх об'єктів за їхніми адресами;
- збереження всіх глобальних визначень у глобальній таблиці символів, яка тепер відображатиме визначення об'єктів на їх остаточні віртуальні адреси.

Приклад виконання першого проходу компонувальника показано на рис. 10.2.

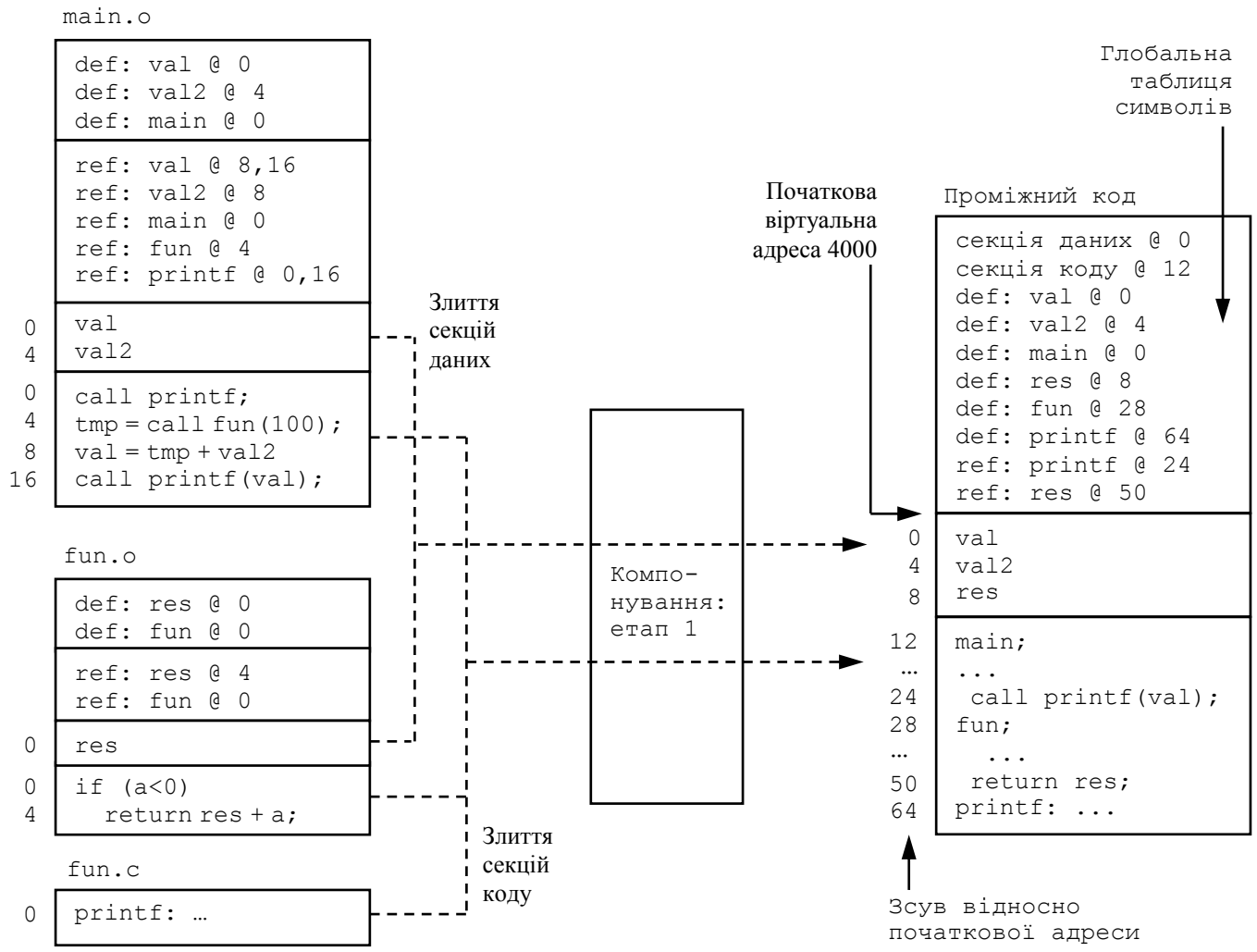

Рис. 10.2. Виконання першого проходу компонувальника

Головне завдання, яке розв'язують на другому проході компонувальника, полягає в корекції всіх адрес в об'єктному коді та розв'язанні всіх зовнішніх посилань. Для цього після збереження всіх посилань у глобальній таблиці символів компонувальник перевіряє, щоб кожний символ мав тільки одне визначення (бути використаний він може скільки завгодно разів). Після цього проходить всіма посиланнями і виконує їхню корекцію, замінюючи посилання адресою відповідного символу.

Процес підстановки адрес замість посилань називають ще налаштуванням адрес (address relocation). Крім основного виду такого налаштування, є інші його варіанти:

- налаштування із базовою адресою і зсувом, коли деякий набір пов'язаних імен, для яких задані зсуви, переводять у набір адрес додаванням зсувів до однієї базової адреси; таке налаштування використовують, наприклад для елементів структур (у сенсі мови С); при цьому звертання до поля структури переводять в адресу додаванням зсуву поля та базової адреси структури;
- корекція статичних даних секцій у разі їхнього переміщення (адреси всіх таких елементів скориговують на величину переміщення). Приклад виконання другого проходу компонувальника показано на рис. 10.3.

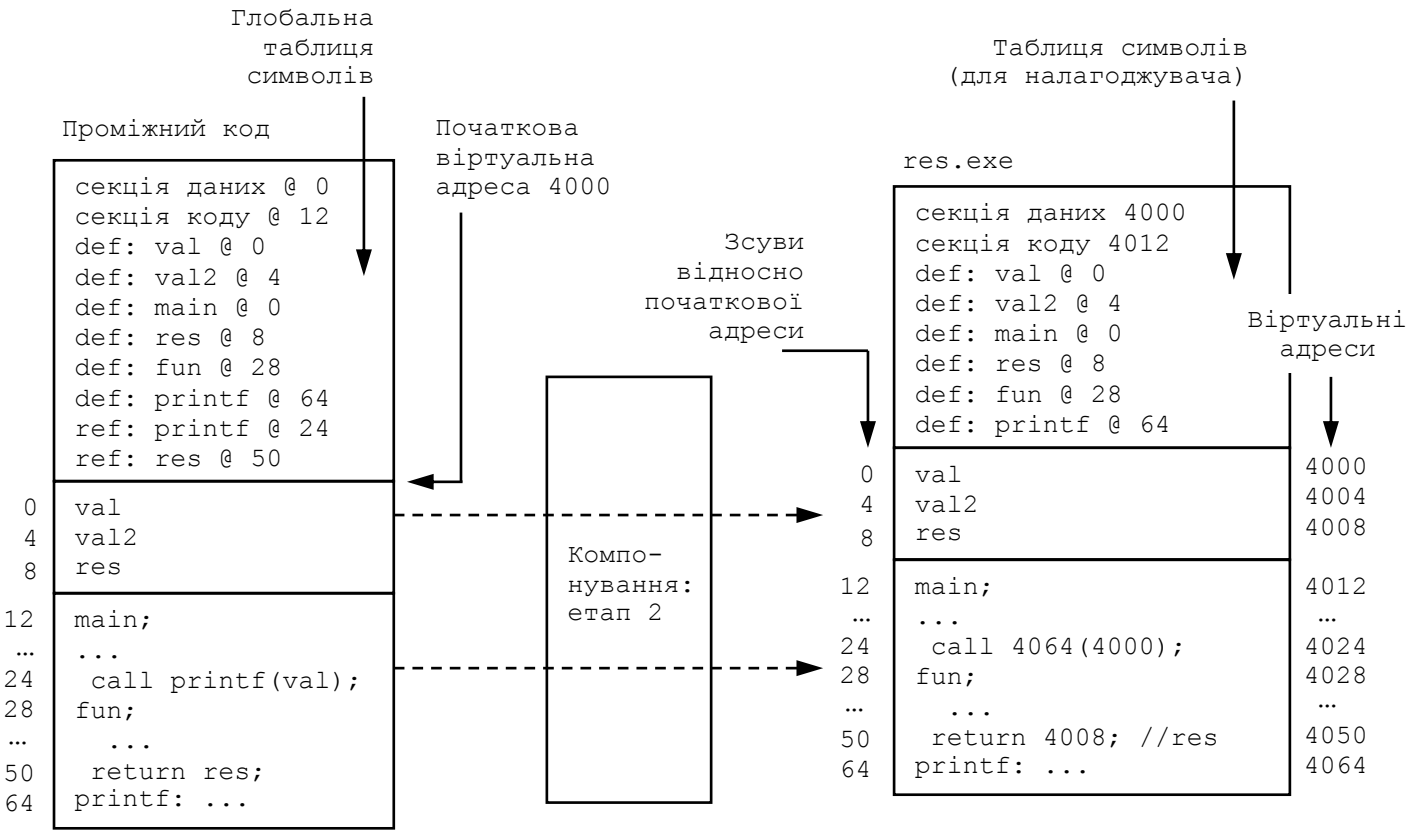

Рис. 10.3. Виконання другого проходу компонувальника

10.3. Завантаження виконуваних файлів за статичного компонування

Виконуваний файл, отриманий внаслідок описаного раніше статичного компонування, містить усе необхідне для створення процесу. Завантаження такого файла у пам'ять виконує окремий компонент ОС – *програмний завантажувач* (program loader). Він звичайно відображає виконуваний файл в адресний простір процесу (здебільшого файл відображають не як єдине ціле, а секціями, причому ділянки пам'яті для секцій виділяє також завантажувач) та ініціалізує керуючий блок процесу таким чином, щоб процес був у стані готовності до виконання.

Найважливішим етапом є відображення виконуваного файла. Використання відображуваної пам'яті для завантаження коду і даних дає змогу реалізувати низку технологій оптимізації такого завантаження.

- Так забезпечують *завантаження на вимогу* (demand loading), коли код не завантажують із диска до його виконання. Під час відображення виконуваного файла в адресний простір процесу спроба виконання коду, не завантаженого у пам'ять, спричинить автоматичне завантаження нової сторінки внаслідок сторінкового переривання.
- Секція даних може виявитися розрідженою і дані не зчитуватимуться із диска, замість них генеруватимуться сторінки, заповнені нулями.
- Так полегшують реалізацію спільного використання сторінок коду кількома процесами.
- Якщо кеш файлової системи використовує відображувану пам'ять, сторінки виконуваного файла після завантаження потраплятимуть у цей кеш.

Зазначимо, що під час відображення виконуваного файла у пам'яті автоматично розміщають його код та ініціалізовані дані. Стек і динамічну ділянку пам'яті зазвичай створюють заново, при цьому для динамічної ділянки компілятор і компонувальник можуть тільки задати її початок, а всю інформацію із керування стеком визначає компілятор із використанням адресації щодо покажчика стека (який буде встановлено завантажувачем).

10.4. Динамічне компонування

Поняття динамічної бібліотеки

Статичне компонування виконуваних файлів має низку недоліків.

- Якщо кілька застосувань використовують спільний код (наприклад, код бібліотеки мови С), кожний виконуваний файл міститиме окрему копію цього коду; у результаті такі файли займатимуть значне місце на диску і у пам'яті. Хотілося б мати можливість зберігати на диску і завантажувати у пам'ять тільки одну копію спільного коду.
- Під час кожного оновлення застосування потрібно перекомпілювати, перекомпонувати і перевстановити.
- Неможливо реалізувати динамічне завантаження програмного коду під час виконання (наприклад, якщо потрібно реалізувати модульну структуру програми, подібно до того, як це зроблено у ядрі Linux).

Для вирішення цих і подібних проблем було запропоновано концепцію динамічного компонування із використанням динамічних або розділюваних бібліотек (dynamic-link libraries (DLL), shared libraries).

Динамічна бібліотека – набір функцій, скомпонованих разом у вигляді бінарного файла, який може бути динамічно завантажений в адресний простір процесу, що використовує ці функції. *Динамічне завантаження* (dynamic loading) – завантаження під час виконання процесу (зазвичай реалізоване як відображення файла бібліотеки в його адресний простір), *динамічне компонування* (dynamic linking) – компонування образу виконуваного файла під час виконання процесу із використанням динамічних бібліотек.

Переваги і недоліки використання динамічних бібліотек

Переваги використання динамічних бібліотек (для стислості вживатимемо також термін «DLL», який частіше застосовують для позначення динамічних бібліотек у Windows-системах) наведено нижче.

- Оскільки бібліотечні функції містяться в окремому файлі, розмір виконуваного файла стає меншим. Якщо врахувати, що є динамічні бібліотеки, які використовують майже всі застосування у системі (стандартна бібліотека мови С у

Linux, бібліотека підсистеми Win32 у Windows XP), то очевидно, що так заощаджують дуже багато дискового простору.

- Якщо динамічну бібліотеку використовують кілька процесів, у пам'ять завантажують лише одну її копію, після чого сторінки коду бібліотеки відображаються в адресний простір кожного з цих процесів. Це дає змогу ефективніше використовувати пам'ять.
- Оновлення застосування може бути зведене до встановлення нової версії динамічної бібліотеки без необхідності перекомпонування тих його частин, які не змінилися.
- Динамічні бібліотеки дають змогу застосуванню реалізувати динамічне завантаження модулів на вимогу. На базі цього може бути реалізований розширюваний API застосування. Для додавання нових функцій до такого API стороннім розробникам достатньо буде створити і встановити нову динамічну бібліотеку, яка підлягає певним правилам.
- Динамічні бібліотеки дають можливість спільно використовувати ресурси застосування (наприклад, така бібліотека може містити спільний набір піктограм), крім того, вони дають змогу спростити локалізацію застосування (якщо всі рядки, які використовуються програмою, помістити в окрему DLL, для заміни мови застосування достатньо буде замінити тільки цю DLL).
- Оскільки динамічні бібліотеки є двійковими файлами, можна організувати спільну роботу бібліотек, розроблених із використанням різних мов програмування і програмних засобів, що спрощує створення застосувань на основі програмних компонентів (отже, динамічне компонування лежить в основі компонентного підходу до розробки програмного забезпечення).

Динамічні бібліотеки не позбавлені ***недоліків*** (хоча вони тільки в окремих випадках виправдовують використання статичного компонування).

- Використання DLL сповільнює завантаження застосування. Що більше таких бібліотек потрібно процесу, то більше файлів треба йому відобразити у свій адресний простір під час завантаження, а відображення кожного файла забирає час. Для прискорення завантаження рекомендують укрупнювати DLL, об'єднуючи кілька взаємозалежних бібліотек в одну загальну.
- У деяких ситуаціях (наприклад, під час аварійного завантаження системи із дискети) використання спільних системних DLL неприйнятне через нестачу дискового простору для їхнього зберігання (такі системні DLL, подібно до стандартної бібліотеки мови C, можуть займати кілька мегабайтів дискового простору, при цьому застосування часто потребують усього по кілька функцій із них). У такій ситуації найчастіше використовують версії застосувань, статично скомпоновані таким чином, щоб у їхні виконувані файли був включений зі стандартних бібліотек код лише тих функцій, які їм потрібні.
- Найбільшою проблемою у використанні динамічного компонування є проблема зворотної сумісності динамічних бібліотек. Розглянемо її окремо.

Зворотна сумісність динамічних бібліотек

Ця проблема виникає в ситуації, коли застосування встановлює нову версію DLL поверх попередньої. Якщо нова версія не має зворотної сумісності із попередніми, застосування, розраховані на використання попередніх версій бібліотеки, можуть припинити роботу. Досягти такої сумісності досить складно, особливо коли попередня версія містила відомі помилки, і застосування, що використовують бібліотеку, розробили код їхнього

обходу – виправлення помилки у бібліотеці може зробити код застосування невірним (кажуть, що в цьому разі порушується сумісність за помилками – bug-to-bug compatibility).

Деякі ОС (переважно це стосується Windows-систем, але певні проблеми є й в UNIX) ускладнювали цю проблему через те, що не давали можливості кільком версіям однієї й тієї самої бібліотеки одночасно бути завантаженими у пам'ять; виділяли для динамічних бібліотек усіх застосувань спільний каталог, цим «запрошуючи» застосування перезаписувати динамічні бібліотеки один одного (можлива була навіть ситуація, коли стару версію бібліотеки записували поверх нової); не зберігали в динамічних бібліотеках і застосуваннях інформації про точні версії бібліотек, від яких вони залежать.

Усе це призвело до ситуації, котра стосовно Windows-систем (насамперед Consumer Windows) дістала назву «пекло DLL» (DLL hell), коли із часом виявлялося неможливо визначити, що за версія бібліотеки була встановлена і яким застосуванням і що за версія і якому застосуванню потрібна насправді. По суті не було способу гарантовано забезпечити використання застосуванням тієї версії бібліотеки, з розрахунком на яку воно розроблялося.

Розроблювачі сучасних ОС намагаються виправити цю ситуацію (блокуванням і резервним копіюванням важливих DLL, дозволом кільком версіям DLL бути одночасно завантаженими у пам'ять, а також підтримкою використання застосуванням тільки бібліотек із його робочого каталогу).

Неявне і явне зв'язування

Є два основні способи завантаження динамічних бібліотек в адресний простір процесу – *неявне* і *явне зв'язування* (implicit і explicit binding).

Неявне – основний спосіб завантаження динамічних бібліотек у сучасних ОС. При цьому бібліотеку завантажують автоматично до початку виконання застосування під час завантаження виконуваного файла, за це відповідає завантажувач виконуваних файлів ОС. У деяких системах такий завантажувач є частиною ядра ОС, у деяких – окремим застосуванням. Список бібліотек, потрібних для завантаження, зберігають у виконуваному файлі. До переваг цього методу належать:

- простота і прозорість з погляду програміста (йому не потрібно писати код завантаження бібліотек, достатньо у налаштуваннях компонувальника вказати список бібліотек, які йому потрібні);
- висока ефективність роботи процесу після початкового завантаження (усі необхідні бібліотеки до цього часу вже завантажені у його адресний простір).

Недоліком неявного зв'язування можна вважати зниження гнучкості (так, наприклад, якщо хоча б однієї з необхідних бібліотек не буде на місці, процес завантаження не обійдеться без проблем, навіть коли для виконання конкретної задачі ця бібліотека не потрібна). Крім того, збільшуються час завантаження і початковий обсяг необхідної пам'яті.

Альтернативним для неявного є явне зв'язування, коли динамічну бібліотеку завантажують в адресний простір процесу виконанням системного виклику із його коду. Після цього, використовуючи інший системний виклик, застосування отримує адресу необхідної йому функції бібліотеки і може її викликати. Після використання бібліотеку можна вилучити з пам'яті. Компонувальник при цьому нічого про неї не знає, завантажувач ОС автоматично бібліотек не завантажує (здебільшого неявне зв'язування зводиться до автоматичного виконання тих самих викликів, які сам програміст виконує за явного).

Такий підхід вимагає від програміста додаткових зусиль, але має більшу гнучкість. Однак складність реалізації призводить до того, що його використовують лише тоді, коли застосуванню справді потрібно завантажувати і вивантажувати додаткові бібліотеки під час виконання.

Динамічні бібліотеки та адресний простір процесу

Після того, як динамічна бібліотека була відображена в адресний простір процесу, вона стає майже прозорою для програмного коду, що тут виконується.

Усі функції бібліотеки стають доступними для всіх потоків цього процесу, фактично її код і дані набувають вигляду доданих до адресного простору процесу. Зазначимо, що під час відображення бібліотеки у пам'ять використовують технологію копіювання під час записування, тому кожен процес матиме свою копію стека і даних бібліотеки.

З іншого боку, для коду бібліотечної функції будуть доступні такі ресурси, як дескриптори відкритих файлів процесу і стек потоку, що викликав дану функцію. Не слід, однак, забувати про те, що під час роботи із даними потоку із коду бібліотеки потрібно виявляти обережність, зокрема, ніколи не вивільняти пам'ять, розподілену не в цій бібліотеці. Іншими словами, код бібліотек, розрахованих на використання у багатопотокових застосуваннях, має бути безпечним з погляду потоків (thread-safe).

Особливості об'єктного коду динамічних бібліотек

Код динамічних бібліотек звичайно зберігають у виконуваних файлах формату, стандартного для цієї ОС (далі побачимо, що виконувані файли і файли DLL можуть відрізнятися тільки одним бітом у заголовку), але з погляду характеру цього коду є одна важлива відмінність між кодом DLL і кодом звичайних виконуваних файлів. Вона полягає в тому, що код DLL в один і той самий час повинен мати можливість завантажуватися за різними адресами. Для того щоб це було можливо, такий код потрібно робити позиційно-незалежним.

Позиційно-незалежний код завжди використовує відносну адресацію (базову адресу додають до зсуву). Базову адресу налаштовують у момент завантаження DLL в адресний простір процесу і називають також базовою адресою бібліотеки; такі адреси відрізняються для різних процесів. Зсув у цьому разі називають внутрішнім зсувом об'єкта.

Точка входу динамічної бібліотеки

Одна із функцій динамічної бібліотеки може бути позначена як її *точка входу.* Така функція автоматично виконуватиметься завжди, коли цю DLL відображають в адресний простір процесу (явно або неявне); у неї можна поміщати код ініціалізації структур даних бібліотеки. Багато систем дають змогу задавати також і функцію, що викличеться в разі вивантаження DLL із пам'яті.

10.5. Структура виконуваних файлів

У сучасних ОС є тенденція до спрощення процедури завантаження виконуваних файлів через наближення їхнього формату до образу процесу у пам'яті. Описати загальну структуру виконуваного файлу доволі складно, у цьому розділі зупинимося на деяких спільних компонентах таких файлів, а потім розглянемо конкретні формати.

Оскільки виконувані файли створює компонувальник на базі об'єктних файлів, то у структурі цих файлів є багато спільного. Насамперед це стосується того, що обидва види файлів складаються з набору секцій різного призначення.

Деякі спільні елементи структури виконуваних файлів (їхній порядок і точний зміст розрізняють для різних форматів цих файлів) наведено нижче.

- Насамперед виконувані файли мають заголовок. Він найчастіше містить «магічні символи», які дають змогу ОС швидко визначити, що цей файл є виконуваним конкретного типу; базову адресу відображення виконуваного коду у пам'ять; ознаку відмінності між незалежним виконуваним файлом і DLL; адреси найважливіших елементів файла.

- Майже завжди в такі файли включають інформацію для динамічного компонувальника. Звичайно ця інформація складається зі *списку імпорту,* що містить інформацію про всі DLL, потрібні для виконання цього файла (для неявного зв'язування) та *списку експорту,* що містить інформацію про всі функції, доступні для використання іншими виконуваними файлами.
- Деякі формати виконуваних файлів використовують зовнішній динамічний завантажувач; у цьому разі всередині файла також зберігають інформацію про його місцезнаходження.
- Інформацію про всі секції файла зберігають у списку секцій, який називають таблицею секцій (section table). Елементи цієї таблиці описують різні секції файла.
- Нарешті, у файлі розташовані самі секції, що містять дані різного призначення. Назви секцій та їхній вміст розрізняються для різних форматів, але майже завжди є секції для коду та ініціалізованих даних.

10.6. Виконувані файли в Linux

Формат ELF

Формат ELF є основним форматом виконуваних файлів для Linux та інших сучасних UNIX-систем. Цей формат можна використати для таких типів файлів:

- об'єктних, призначених для статичного компонування (їх також називають переміщуваними файлами);
- виконуваних, котрі описують, яким чином виклик `exec()` створює образ процесу;
- розділюваних (динамічних) бібліотек, які динамічний компонувальник збирає в образ процесу.

Загальну структуру файла у форматі ELF показано на рис. 10.4.

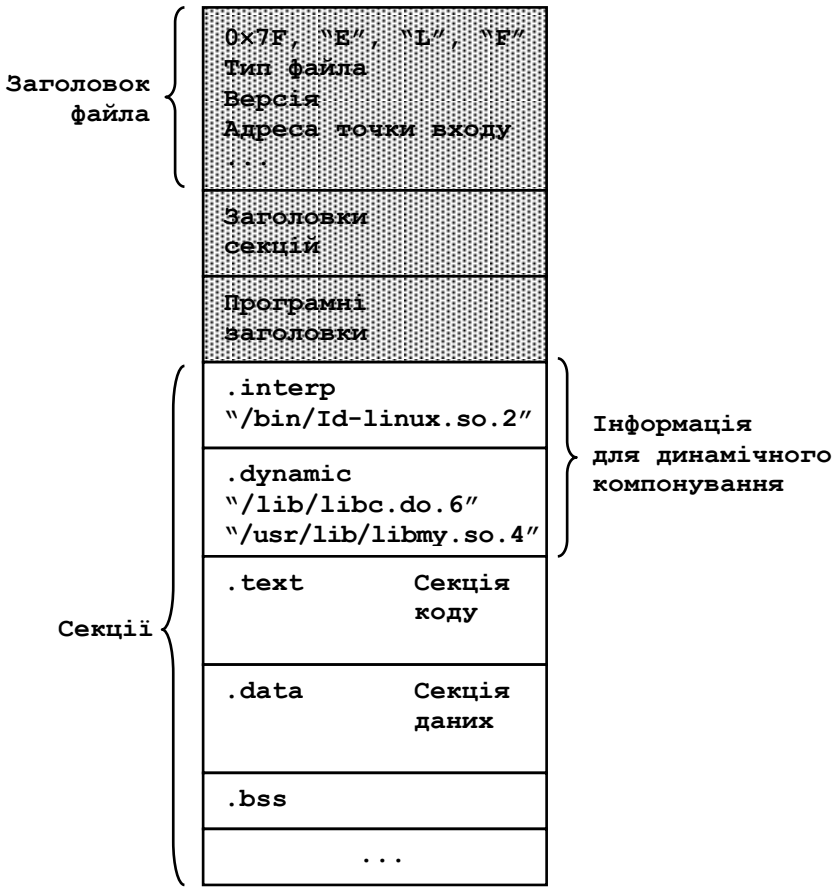

Рис. 10.4. Структура файла в ELF форматі

Заголовок файла

Заголовок файла у форматі ELF описує його організацію. Він починається із чотирьох обов'язкових «магічних» символів O×7F, 'E', 'L', 'T'; крім цього, містить інформацію про тип файла (об'єктний файл, виконуваний файл, динамічна бібліотека), архітектуру процесора, версію файла, адресу точки входу (за яким ОС передасть керування після завантаження), нарешті, про зсув у файлі таблиці програмних заголовків і таблиці заголовків секцій (ці таблиці розглянемо окремо).

Секції файла

Інформація про всі секції ELF-файла перебуває у *таблиці заголовків секцій.* Вона містить заголовки секцій, кожен із яких описує одну секцію і включає її ім'я, тип, розмір, адресу, на яку вона має відображатися, інформацію про те, чи дозволене записування у секцію після її завантаження у пам'ять.

Розглянемо деякі наперед визначені секції ELF-файла: `.text` містить виконуваний код програми, `.data` і `.data1` – ініціалізовані дані; `.bss` призначена для організації сегмента неініціалізованих даних у пам'яті; вона реалізована як «діра» у розрідженому файлі, після завантаження якого у пам'ять спроба звернутися до даних секції спричиняє повернення сторінки, заповненої нулями; `.symtab` містить таблицю символів цього файла; `.strtab` – таблицю рядків цього файла, в якій розташовані використовувані в коді програми рядкові константи.

Об'єктні файли містять також спеціальну секцію `.reloc` із даними для налаштування адрес. Крім цього, можна виокремити кілька секцій, що зберігають інформацію для системного завантажувача і динамічного компонувальника, що буде розглянуто окремо. Застосування можуть задавати і свої власні секції.

Зазначимо, що в термінології ELF секція означає поіменований розділ у виконуваному файлі; після відображення у пам'ять секції відповідає сегмент.

Програмний заголовок

ELF-файли містять інформацію, яка буде використана для створення образу процесу після завантаження цього файла у пам'ять. Кажуть про два відображення інформації у форматі ELF – як дискового файла і як образу процесу. Структурою даних, що керує завантаженням ELF-файлів, є програмний заголовок (program header).

Після завантаження у пам'ять на базі секцій ELF-файлу створюють сегменти пам'яті, причому один сегмент може відповідати кільком секціям. Програмний заголовок – це масив елементів, кожен із яких описує один сегмент. Такий елемент може містити тип сегмента (код, дані тощо), його зсув від початку файла, віртуальну адресу початку сегмента в пам'яті, розмір сегмента на диску і в пам'яті. Інформацію про динамічне компонування завантажують в особливі сегменти.

Базову адресу бібліотеки або виконуваного файла обчислюють на основі адреси завантаження та віртуальної адреси першого завантаженого сегмента, визначеного в програмному заголовку (при цьому його округляють до розміру сторінки).

Динамічне компонування в Linux

Спочатку розглянемо елементи ELF-файла, відповідальні за динамічне компонування, а потім перейдемо до опису всього процесу такого компонування бібліотек формату ELF.

Структури даних підтримки динамічного компонування

Для того щоб забезпечити динамічне компонування, ELF-файл містить ряд структур даних. Розглянемо деякі з них.

- Шлях до динамічного компонувальника задають тому, що ELF-формат передбачає реалізацію такого компонувальника окремою програмою. Цей шлях розміщають у спеціальній секції `.interp` і завантажують в особливий сегмент під час виконання. Про принципи роботи такого компонувальника йтиметься окремо.
- Інформацію про необхідні динамічні бібліотеки зберігають у спеціальній секції `.dynamic`, там також міститься інформація про точку входу динамічної бібліотеки.

Зазначимо, що в ELF-файлі для динамічних бібліотек не задають окремої інформації про набір функцій, експортованих цією бібліотекою, вважаючи, що бібліотека експортує всі функції, інформація про які наявна у її таблиці символів.

Імена і розташування динамічних бібліотек

До розділюваних бібліотек можна звертатися в різних ситуаціях за трьома різними іменами.

- *Основним* (soname). Використовується динамічним компонувальником і має таку структуру: `libname.so.N`, де `name` – ім'я бібліотеки, `N` – базовий номер версії (зміна цього номера зазвичай пов'язана із несумісними змінами в інтерфейсі бібліотеки). Повне основне ім'я включає каталог, у якому перебуває бібліотека. Приклад основного імені: `libdl.so.1`.
- *Реальним* (real name). Це ім'я файла, у якому зберігається виконуваний код бібліотеки. Воно додає до основного імені `.M1.M2`, де `M1` – молодший номер версії, `M2` – номер реалізації (останній можна пропускати). Прикладом реального імені може бути `libdl.so.1.9.5`. Повне основне ім'я задають як символічний зв'язок, що вказує на цей файл.
- *Для компонувальника* (linker name). Використовують під час компонування застосування, що потребує бібліотеки; воно – основне ім'я без номера версії: `libdl.so`. Таке ім'я задають як символічний зв'язок, що вказує на повне основне ім'я. Компонувальник отримує ім'я `libxx.so` у вигляді параметра `-lxx`, для `libdl.so` цей параметр матиме такий вигляд: `-ldl`.

Ось фрагмент відображення вмісту каталогу, на якому видно всі три імені бібліотеки:

```
lrwxrwxrwx ... libdl.so -> libdl.so.1*
lrwxrwxrwx ... libdl.so.1 -> libdl.so.1.9.5*
-rwxr-xr-x ... libdl.so.1.9.5*
```

Під час розробки бібліотеки створюють файл із реальним іменем. Коли встановлюють нову версію динамічної бібліотеки, його поміщають в один із наперед визначених каталогів, після чого запускають спеціальну утиліту `ldconfig`, яка перевіряє наявні файли й автоматично створює символічні зв'язки для основних імен на підставі інформації про версії з ELF-заголовка.

Використання динамічних бібліотек із застосувань

Коли компонують застосування, що використовує динамічну бібліотеку, необхідно вказати ім'я для компонувальника, яке посилається на основне ім'я. Основне ім'я збережеться у виконуваному файлі застосування у списку необхідних бібліотек (у секції

.dynamic). Жодних додаткових дій у коді застосування виконувати не потрібно – усі функції бібліотеки будуть доступні в ньому як глобальні функції. Ось приклад запуску компілятора gcc для збирання застосування, яке використовує бібліотеку з іменем для компонувальника libabc.so, що перебуває в каталозі /usr/local/lib:

```
$ gcc myprog.c -o myprog -labc -L /usr/local/lib
```

Розробка динамічних бібліотек

Динамічні бібліотеки в Linux створюють за допомогою стандартного компілятора мови С, подібно до будь-якого виконуваного файла. Компонувальнику треба вказати на необхідність генерації позиційно-незалежного коду (для gcc це параметр – fpic), а також задати ім'я, яке потрібно використати як основне ім'я бібліотеки (soname). Для gcc його треба задавати так: -Wl, -soname, основне_ім'я.

Наведемо приклад запуску компілятора gcc для збирання файла бібліотеки libabc.so.1.0.0 з основним іменем libabc.so.1:

```
$ gcc -shared -o libabc.so.1.0.0 -fpic -Wl.-soname.libabc.so.1 abc.c
```

Розробка коду бібліотеки не потребує додаткових дій – усі створені глобальні функції будуть доступні для застосувань, що використовують бібліотеку.

Точка входу у бібліотеку має реалізовуватись як функція _init(). Її викликатимуть щоразу під час відображення бібліотеки у пам'ять процесу. Функція, яку викликатимуть у разі вивантаження бібліотеки із пам'яті, називається _fini().

Динамічний компонувальник і неявне зв'язування бібліотек

Запуск виконуваного файла у форматі ELF призводить до того, що керування отримує *динамічний компонувальник,* шлях до якого зазначений у секції .interp ELF-файла. У Linux такий компонувальник називають /lib/ld-linux.so.N (N – номер версії). Він у свою чергу відшукує і завантажує всі динамічні бібліотеки, потрібні для виконання застосування, використовуючи список необхідних бібліотек (секцію .dynamic); під час пошуку він використовує основні імена. Так реалізоване неявне зв'язування.

Список каталогів, у яких динамічний компонувальник має шукати бібліотеки, задають у його конфігураційному файлі /etc/ld.so.conf. Стандартними каталогами для динамічних бібліотек є /lib і /usr/lib, пошук у них здійснюють, навіть якщо не заданий конфігураційний файл.

Пошук у всіх цих каталогах під час кожного запуску застосування неефективний, тому організовують кеш імен динамічних бібліотек і зберігають у файлі /etc/ld.so.cache. Його створює утиліта ldconfig після того, як встановить усі символічні зв'язки. У разі необхідності шлях до потрібної бібліотеки вибирають із кеша, що підвищує ефективність завантаження застосувань. На жаль, підтримка коректного стану кеша вимагає ручного запуску ldconfig під час операцій додавання, вилучення або зміни будь-якої динамічної бібліотеки, інакше динамічний компонувальник не врахує цих змін. Це видається недостатньо зручним.

Можна змусити застосування використати для конкретного запуску іншу версію динамічної бібліотеки. Для цього використовують змінну оточення LD_LIBRARY_PATH, що є списком каталогів. За наявності такої змінної саме із каталогів цього списку починає пошук бібліотек динамічний компонувальник. Якщо в одному із каталогів цього списку буде розміщена інша версія бібліотеки, саме вона буде використана замість стандартної версії, наприклад з /usr/lib.

Ця змінна зручна також тоді, коли зміна у бібліотеці, що не зачіпає основного імені,

призвела до порушення роботи деякого застосування. Для цього потрібно перенести стару версію бібліотеки в інший каталог і задати перед запуском застосування значення LD_LIBRARY_PATH, що включає каталог зі старою версією.

Дізнатися, які динамічні бібліотеки потрібні застосуванню, можна за допомогою утиліти `ldd`, параметром якої є ім'я виконуваного файла.

Явне зв'язування динамічних бібліотек

Для явного завантаження бібліотеки під час виконання застосування та виклику її функцій потрібно виконати кілька кроків.

1. Завантажити бібліотеку за допомогою системного виклику `dlopen(libpath, flags)`. Як параметри цей виклик приймає:
   - шлях до бібліотеки `libpath` (у разі задавання повного шляху виклик відразу знаходить файл, якщо задано тільки ім'я – здійснює пошук, аналогічний до того, що виконує динамічний компонувальник);
   - набір прапорців `flags`, які керують розв'язанням посилань під час завантаження (наприклад, вмикання прапорця RTLD_NOW означає, що всі зовнішні посилання мають розв'язуватися під час завантаження і в разі відмови у розв'язанні хоча б одного посилання завантаження не відбудеться).

   Цей виклик повертає дескриптор бібліотеки, який використовується в інших функціях.
2. Знайти символ у бібліотеці за іменем шляхом виклику `dlsym(libd, sym)`. Першим параметром цього виклику є дескриптор бібліотеки, другим – ім'я символу. Цей виклик повертає адресу символу (функції) у бібліотеці, через цей покажчик можна викликати бібліотечну функцію.
3. Закрити дескриптор бібліотеки за допомогою виклику `dlclose(libd)`. Бібліотеку вивантажать із пам'яті, якщо для неї не залишиться жодного відкритого дескриптора.

   Ось приклад завантаження бібліотеки і виклику функції з неї:

```
#include <dlfcn.h>
typedef int(*fint)(int);
fint fun;    // оголошення покажчика на функцію
void *libd = dlopen("libmy.so.1", RTLD_NOW);
if (libd) {
     fun = dlsym(libd, "fun");
     int res = (*fun)(100); // виклик функції через покажчик
     dlclose(libd);
}
```

Автоматичний виклик інтерпретаторів

Під скриптами звичайно розуміють програми, написані на різних інтерпретованих мовах – Perl, Python, TCL, мові командного інтерпретатора UNIX (shell-скрипти). Звичайний спосіб запуску таких скриптів потребує явного задавання імені інтерпретатора в командному рядку

```
$ perl test.pl
```

У цьому разі буде викликано інтерпретатор мови Perl, а йому на вхід подано Perl-скрипт `test.pl`, після чого інтерпретатор починає виконувати цей скрипт. Зазначати інтерпретатор щоразу під час запуску скрипта не зовсім зручно.

У всіх UNIX-системах є спосіб виконувати такі файли без вказування імені інтерпретатора в командному рядку. Ця можливість реалізована на рівні завантажувача виконуваних файлів.

Для того, щоб файл скрипта був поданий завантажувачем виконуваних файлів на вхід інтерпретатору автоматично, у першому рядку цього файла треба зазначити повний шлях до інтерпретатора після комбінації символів #! і, крім того, для цього файла задати права на виконання.

Ось приклад скрипта на Perl, яка використовує цю технологію:

```
#!/usr/bin/perl
# далі йде звичайний текст скрипта test.pl
```

Спробу виконання такого файла перехоплює завантажувач, який завантажує у пам'ять заданий інтерпретатор, а як параметр командного рядка передає йому файл, запуск якого він перехопив.

10.7. Виконувані файли у Windows XP
Формат PE

Файли формату PE (Portable Executable) з'явилися разом із Windows NT. Вони мають багато спільного із традиційними форматами UNIX-систем (об'єктні файли, на основі яких будують такий файл, можуть бути у форматі COFF – одному зі стандартних форматів об'єктних файлів у UNIX). Загальна структура файла в форматі PE наведена на рис. 10.5.

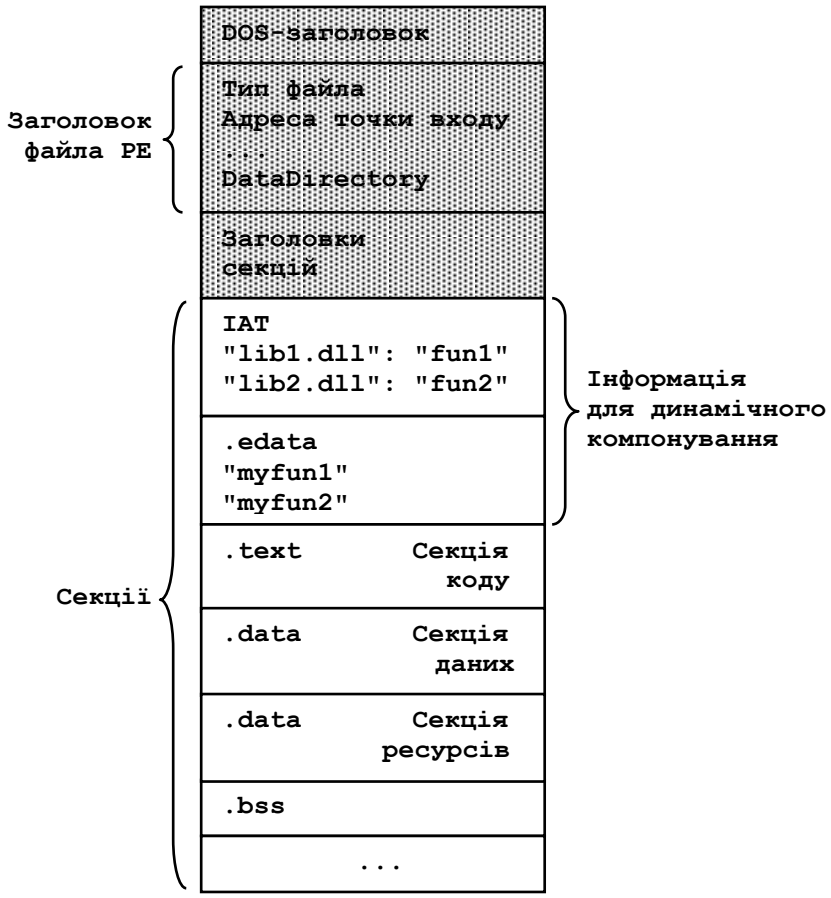

Рис. 10.5. Структура файла у форматі PE

Основна подібність між двома форматами полягає у їхній секційній організації та описі списку секцій таблицею секцій. Зазначимо, що навіть назви і призначення багатьох основних секцій збігаються – це стосується `.text`, `.data`, `.bss`, `.reloc`. Окрема секція, як і в ELF, виділена під опис списку імпорту.

Основні відмінності між двома форматами наведені нижче.

- Перед основним заголовком PE-файла поміщають так званий DOS-заголовок. Він містить невелику програму, яка у разі запуску під MS-DOS виводить повідомлення і завершується. Цей заголовок залишився відтоді, коли PE-файли часто намагалися запускати під керуванням MS-DOS або Windows 3.x. Під час звичайного запуску (під Windows XP) керування негайно передають за адресою початку виконання, заданою в основному заголовку.
- Інформацію, необхідну для виконання файла, не виносять в окрему структуру даних, а зберігають у стандартних структурах – основному заголовку, заголовках секцій тощо.
- Для зручності доступу адреси найважливіших об'єктів усередині файла (секцій експорту, імпорту, ресурсів тощо) утримують у заголовку файла як окремий масив DataDirectory.
- Окрема секція `.rsrc` зарезервована для зберігання ресурсів програми. Ресурси всередині цієї секції мають деревоподібну організацію із каталогами і підкаталогами.

Формат PE і динамічне компонування у Windows XP

Спочатку розглянемо відмінності підтримки динамічного компонування у форматі PE від підтримки формату ELF.

- У Windows XP динамічне компонування здійснює система, а не окремий динамічний компонувальник; очевидно, що всередині виконуваного файла ім'я такого компонувальника не задають.
- Окрема секція `.edata` виділена для опису експортованих символів. Це досить важлива відмінність від ELF, для якого всі символи за замовчуванням є експортованими.
- Відмінності має і секція імпортованих функцій. Тут створена спеціальна таблиця, що визначає всі імпортовані функції; після запуску її заповнюють адресами функцій з DLL. Таку таблицю називають таблицею імпорту адрес (IAT). Використання IAT дає змогу звести всю інформацію про імпортовані адреси в одне місце у файлі.

Зазначимо, що як і для ELF, відмінностей між динамічними бібліотеками і виконуваними файлами із погляду формату файла немає, фактично їх розрізняють за значенням поля типу в заголовку.

Процес компонування у разі неявного зв'язування

Для реалізації неявного зв'язування все, що потрібно від розробника застосування, – це вказати компонувальнику список необхідних DLL і впевнитися, що під час виконання всі вони можуть бути знайдені. Завантажувач ОС забезпечить пошук і виконання потрібного коду. Зауважимо, що якщо під час завантаження DLL виникла помилка, весь процес завантаження переривається. До початку виконання основного потоку процесу всі необхідні DLL мають бути відображені в його адресний простір. У розробці DLL, на відміну від UNIX, основним завданням програміста є задавання списку експортованих функцій. Цього можна домогтися такими способами:

- створити спеціальний файл із розширенням `.DEF`, у якому перелічити всі такі функції, і передати його компонувальнику;
- у разі використання Visual C++ скористатися спеціальною конструкцією `_declspec(dllexport)`, яку потрібно поміщати перед оголошенням експортованої функції:

```
_declspec(dllexport) DWORD fun() {
```

```
    Printf("Виклик fun()\n");
    return 100; }
```

При цьому додатково до створення DLL компонувальник згенерує спеціальний файл заглушок – статичну бібліотеку із розширенням .LIB, яку компонують із клієнтським застосуванням і яка містить код заглушок для створення зв'язків із DLL під час завантаження. Ім'я цієї бібліотеки має бути явно задане як один із параметрів виклику компонувальника під час компонування клієнтського застосування.

Компонуючи DLL за допомогою Visual C++, потрібно вмикати прапорець `-LD`:

```
c:\mydll> ci -LD -o mydll.dll mydll.c
```

У разі використання функцій із DLL їх, на відміну від UNIX, потрібно імпортувати. Це може мати такий вигляд:

```
_declspec(dllimport) DWORD fun();
void main() {
     printf("%d\n", fun());
}
```

Тоді під час компонування будуть узяті функції із .LIB-файла, а в разі виконання заглушки звертатимуться до справжніх функцій із DLL.

Виклик компілятора Visual C++ для компонування клієнтського застосування, що використовує DLL, матиме вигляд:

```
c:\dllclient> cl -o dllclient dllclient.c c:\rnydll\mydll.lib
```

Точка входу в DLL

Точку входу у DLL у Win32 API описують як функцію:

```
BOOL WINAPI DllMain(HINSTANCE libh, DWORD reason, LPVOID reserved);
```

Першим параметром для неї є дескриптор екземпляра бібліотеки, другим – індикатор причини виклику (DLL_PROCESS_ATTACH – під час завантаження бібліотеки, DLL_PROCESS_DETACH – у разі її вивантаження), третій параметр не використовують.

```
 BOOL WINAPI DllMain(HINSTANCE libh, DWORD reason, LPVOID reserved) {
   switch (reason) {
      case DLL_PROCESS_ATTACH; printf("завантаження DLL\n"); break;
      case DLL_PROCESS_DETACH; printf("вивантаження DLL\n"); break;
   }
   return TRUE;
}
```

Відкладене завантаження DLL

Для прискорення завантаження застосувань Windows XP надає можливість відкладеного завантаження DLL. У цьому разі бібліотеку пов'язують із виконуваним файлом неявне, але завантажують у пам'ять тільки під час першого звертання до одного із символів, визначених у ній. Для того щоб задати відкладене завантаження для DLL під час компонування клієнта, потрібно вказати її ім'я як аргумент параметра компонувальника DelayLoad; крім того, необхідно скомпонувати із проектом бібліотеку `delayimp.lib`.

```
C:\dllclient>cl -o dllclient dllclient.c c:\mydll\mydll.lib
     delayimp.lib -link -DelayLoad:mydll.dll
```

Процес розробки самої бібліотеки залишається незмінним.

Явне зв'язування

Процес явного зв'язування DLL у Windows XP складається переважно з тих самих кроків, що і для Linux.

1. Для того щоб відобразити DLL в адресний простір процесу, використовують функцію `LoadLibrary(libpath)` з одним параметром, який задає шлях до файла бібліотеки. Ця функція повертає дескриптор екземпляра бібліотеки (значення типу HINSTANCE). Вона є аналогом `dlopen()`.
2. Аналогом `dlsym()` для отримання покажчика на функцію за її іменем є функція `GetProcAddress(libh, sym)`. Її параметри за змістом ті самі, що і для `dlsym()` – дескриптор екземпляра і рядок з іменем функції.
3. Для вивантаження бібліотеки із пам'яті використовують функцію `FreeLibrary(libh)`.

Ось приклад застосування явного зв'язування у Win32 API (він майже нічим не відрізняється від прикладу для Linux):

```
typedef int(*fint)(int);
fint fun;
HINSTANCE libh = LoadLibrary("my.dll");
if (libh) {
      fun = (fint) GetProcAddress(libh."fun");
      int res = fun(100);
      FreeLibrary(libh);
}
```

Зворотна сумісність DLL у Windows 2000 і Windows XP

У цьому розділі розглянемо, як вирішують проблему зворотної сумісності динамічних бібліотек в останніх версіях ОС лінії Windows XP.

Переспрямування DLL

Для того щоб вирішити проблему засмічення системних каталогів динамічними бібліотеками застосувань, у Windows 2000 з'явилася можливість змусити завантажувач спочатку переглядати під час пошуку необхідних DLL робочий каталог застосування. Таку можливість називають *переспрямуванням DLL* (DLL redirection), для її реалізації достатньо помістити в робочий каталог застосування файл із іменем, отриманим із імені виконуваного файла додаванням суфікса `.local` (наприклад, `MyApp.exe.local`, якщо виконуваний файл називають `MyApp.exe`). Вміст файла ролі не відіграє.

Ізольовані застосування і паралельні збірки

У Windows XP запропоновано повніше вирішення даної проблеми. У цій системі з'явилася можливість розробляти застосування, залежність яких від зовнішніх компонентів задають явно, – ізольовані застосування (isolated applications).

Опис залежностей ізольованого застосування зберігають у спеціальному файлі маніфесту застосування (`MyApp.exe.manifest`). У ньому перераховані компоненти, від яких залежить це застосування. Кожен такий компонент відображають паралельною збіркою (side-by-side assembly) – набором взаємозалежних ресурсів, описаних файлом маніфесту збірки (assembly manifest). Звичайно збірку відображають окремою DLL.

Кожна збірка має версію, при цьому у Windows XP можливе одночасне завантаження у пам'ять кількох версій однієї й тієї самої збірки. За це відповідає спеціальний

компонент динамічного завантажувача ОС – менеджер паралельного завантаження (side-by-side manager). Для визначення правильності зв'язування використовують інформацію із файла маніфесту застосування. Якщо в маніфесті описана залежність від конкретної версії збірки, завантажують цю версію, інакше – версію за замовчуванням.

Якщо задано конкретну версію збірки, вона не може бути перезаписана іншою версією тієї самої збірки: нова версія буде доступна паралельно зі старою. Цим вирішують проблему зворотної сумісності – гарантують наявність саме тієї версії динамічної бібліотеки, з розрахунком на яку розроблене застосування.

Висновки

- Особливим видом файлів, які використовують в ОС, є виконувані файли, їх створюють компіляцією та компонуванням. При цьому необхідно забезпечити відображення символьних імен, що присутні у вихідному коді, на адреси пам'яті, з якими може працювати процесор після завантаження такого файла у пам'ять. Сучасні компонувальники використовують для цього кілька підходів.
- Розрізняють статичне і динамічне компонування. Динамічне компонування, яке набуло широкого використання у сучасних ОС, пов'язане зі збиранням образу процесу у пам'яті із динамічних бібліотек під час його виконання.
- Сучасні формати виконуваних файлів, такі як ELF у Linux і PE у Windows XP, мають подібну структуру. Ця структура відбиває образ процесу, що полегшує його відображення у пам'ять.

Контрольні запитання та завдання

1. На якому етапі роботи компонувальника (на першому проході, після першого проходу, на другому проході, після другого проходу) розробник може бути повідомлений про такі особливі ситуації:

а) глобальна змінна повторно визначена в декількох об'єктних файлах;

б) програма не може поміститися у віртуальному адресному просторі;

в) визначена глобальна змінна, котру жодного разу не використовували;

г) задане посилання на неіснуючу зовнішню змінну?

2.Чому в сучасних ОС виконувані файли відображають у пам'ять не одним блоком, а секціями?

3.Опишіть, яким чином використання позиційно-незалежного коду спрощує розробку динамічних бібліотек.

4.Як під час компонування виконуваного файла, у якому є звертання до динамічних бібліотек, забезпечити видачу попереджень про нерозв'язані зовнішні посилання?

5.Чому динамічну бібліотеку завжди відображають в адресний простір процесу повністю, хоча з метою економії пам'яті мало б сенс витягати з неї лише ті функції, які використовує процес?

6.Чому в системі, що використовує динамічне компонування, перший виклик функції з динамічної бібліотеки може виконуватися значно довше, ніж наступні?

7.Чому в разі переходу до використання динамічного компонування розмір балансового набору системи може зменшитися? Як зміниться в цьому випадку розмір робочих наборів окремих процесів?

8.Назвіть переваги і недоліки реалізації динамічного компонування в Linux і Windows XP.

9.Розробіть динамічну бібліотеку для Linux і Windows XP, яка міститиме набір функцій із завдання 8 розділу 11. Створіть тестове застосування, що використовує цю бібліотеку.

10.Реалізуйте застосування для Linux і Windows XP, що може бути розширене під час виконання. Інтерфейс модуля розширення задають набором функцій типу void без параметрів. Після запуску застосування видає на екран підказку й очікує введення команди з клавіатури. Можливі такі команди: load ім'я_модуля (завантаження модуля в пам'ять), unload ім'я_модуля (вилучення модуля з пам'яті), call ім'я_функції (виклик функції з модуля). Кожен модуль розширення повинен містити код, який виконується під час його завантаження в пам'ять та вилучення з пам'яті. Якщо під час завантаження модуля буде встановлено, що імена його функцій збігаються з іменами функцій, завантажених раніше в складі іншого модуля, треба видавати повідомлення про помилку.

11.На основі результатів завдання 10 з розділу 3 і завдання 10 з розділу 14 реалізуйте командний інтерпретатор з розширеною функціональністю. Кожен модуль розширення містить набір функцій, що реалізують команди інтерпретатора. Після завантаження модуля в пам'ять реалізовані в ньому команди стають доступними для користувача. Як приклад розробіть такі модулі:

а) cd_module, що містить код команд cd (зміна поточного каталогу) і pwd (відображення імені поточного каталогу);

б) exit_module, що містить код команди exit.