УДК 004.4′422 DOI: 10.18698/2541-8009-2023-7-921

МЕТОДИКА ЛИНКОВКИ ПРОГРАММНЫХ СИСТЕМ С ИСПОЛЬЗОВАНИЕМ СТАТИЧЕСКИХ И ДИНАМИЧЕСКИХ БИБЛИОТЕК

Р.А. Луцук lra20u222@student.bmstu.ru
Д.В. Мельников melnikovdv@student.bmstu.ru
Т.А. Новиков nta20u673@student.bmstu.ru

МГТУ им. Н.Э. Баумана, Москва, Российская Федерация

Аннотация

Рассмотрен заключительный этап процесса компиляции программы, а именно этап линковки, т. е. связывания в единый файл уже скомпилированных фрагментов кода и кода всех остальных библиотеки. Описан механизм работы библиотеки BFD (Binary File Descriptor) и особенности ее использования. Выполнено сравнение статических и динамических библиотек. Работа компоновщика и его взаимодействие с различными объектными файлами рассмотрены на примере GNU linker. Также было рассмотрены преимущества и недостатки статических и динамических библиотек по таким параметрам, как занимаемая память исполняемого файла, а также время линковки.

Ключевые слова

Программирование, программное обеспечение, проектирование, компиляция, линковка, компоновщик, статическая и динамическая библиотеки, сборка проекта

Поступила в редакцию 05.05.2023

Введение. Любое программное обеспечение (ПО) перед тем, как попасть в руки пользователю, проходит свой жизненный цикл, начиная от первоначальной идеи и заканчивая конечным продуктом. Жизненный цикл практически каждого ПО состоит из шести этапов. Первым этапом является планирование. На этом этапе определяют требования к программному продукту, составляют план разработки, определяют сроки и бюджет проекта. Важно, чтобы на этом этапе были определены все требования к ПО, чтобы избежать дополнительных затрат на доработку продукта в будущем. На втором этапе проводят детальный анализ требований к ПО, определяют функциональные и нефункциональные требования, а также анализируют риски. Третий этап — это проектирование. На этом этапе разрабатывают архитектуру ПО, определяют компоненты и модули, а также проектируют пользовательский интерфейс. Важным критерием служит разработка оптимальной архитектуры ПО для обеспечения высокой производительности и удобства использования. Следующий этап — разработка и сборка продукта. Именно здесь осуществляется написание кода в соответствии с выбранной архитектурой на предыдущем этапе, после чего выполняют сборку, т. е. компиляцию, тестиро-

вание и отладку ПО. На пятом этапе проводят различные виды тестирования, включая модульное, интеграционное, системное и приемочное тестирование, при этом должно быть обеспечено высокое качество ПО и его соответствие требованиям. Заключительный этап — это внедрение. Внедрение подразумевает развертывание ПО в рабочей среде, его настройку и передачу конечным пользователям. После развертывания осуществляется техническая поддержка ПО, включая исправление ошибок и обновление продукта [1].

Компиляция является важным процессом на этапе разработки ПО, который позволяет превратить исходный код программы в исполняемый файл. Иными словами, компиляция — это перевод исходного высокоуровневого кода в машинный код. Исполняемый файл представляет собой файл с командами, которые непосредственно может выполнить компьютер. Компиляция позволяет разработчикам создавать качественное и эффективное ПО, которое может быть активировано на различных платформах, при этом все процессы в компиляторах автоматизированы. В свою очередь, компиляция состоит из нескольких этапов, среди которых имеет место линковка. Именно о ней дальше и пойдет речь [2].

Основной проблемой на этапе линковки считается обеспечение наилучшего быстродействия в сочетании с наименьшими затратами дискового пространства. На данный момент не существует автоматического механизма определения наилучшего метода линковки, следовательно, разработчик должен сам выбирать наиболее подходящий способ для конкретной задачи. В связи с этим при разработке сложных программных систем выбор оптимизированного метода линковки значительно ускоряет процесс сборки, отладки и тестирования проекта.

В процессе разработки методики линковки мы опирались на самые разные источники. Например, в [1, 2] подробно рассмотрены процессы компиляции программ и создания ПО в целом. Также в [2] обсуждаются проблемы, связанные с линковкой, такие как конфликты символов и несовместимость архитектуры.

Руководства пользователя [3, 4] представляют интерес при использовании на практике рассмотренных средств сборки программ. Одно из таких средств GNU linker (version 2, GNU general public license, https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html) по своей сути является расширенной версией официальной документации проекта, в которой подробно представлены алгоритмы работы компоновщика.

В [5] хотя и не рассматриваются вопросы сборки программ, дано общее представление о типах данных, используемых программой, о том, как писать программы с использованием библиотек и функций, что необходимо для понимания логики линковки программы. В [6, 7] проанализированы методы контроля качества ПО, которые необходимо применять на всех этапах разработки, в том числе и на этапе компоновки проекта.

Разработанная в статье методика сравнения видов линковки выбрать нужный метод компоновки программы, что упрощает работу разработчикам и тестировщикам ПО, тем самым оптимизируя количество занимаемой памяти, а также ускоряя сборку программы. Основными отличиями методики от ее аналогов являются простота реализации и отсутствие надобности установки дополнительного ПО. Стоит упомянуть, что немаловажным этапом разработки ПО, положительно влияющим на скорость сборки итогового решения, является оптимальное распределение зависимостей между компонентами ПО. Данную задачу можно упростить с помощью smart-технологий визуального моделирования проектного управления [8].

Цель работы — разработка методики линковки программных систем на основе сравнительного анализа статической и динамической линковки, а именно сравнения занимаемой памяти исполняемого файла и времени процесса линковки.

Для достижения поставленной цели были решены задачи по реализации на языке C кода программы с дальнейшим созданием объектного файла и нахождения необходимых параметров. Изучена работа компоновщика GNU linker. Рассмотрены основные аспекты работы линкера, а также использование библиотек BFD.

Анализ процесса компиляции программных систем. Процесс компиляции состоит из нескольких этапов — рис. 1 [1]. Первым является препроцессинг. На этом этапе совершаются макроподстоновки, т. е. происходит вставка содержимого различных файлов, выполняется условная компиляция, удаляются комментарии. Далее происходит сама компиляция — преобразование полученного на прошлом шаге кода в ассемблерный код. Данный код является промежуточным между машинным кодом и кодом, понятным для человека. После идет этап ассемблирования — преобразования кода на языке ассемблер в машинный код, добавляется информация о внешних функциях и символах. Полученный код сохраняется в объектном файле. Конечным этапом является линковка, которая далее будет более подробно рассмотрена. Линковка представляет собой процесс получения исполняемого файла путем объединения нескольких объектных файлов в единый связанный файл. Помимо объединения объектных файлов на этапе линковки компонуются библиотеки, содержащие используемые в программе команды. Такие библиотеки могут быть статическими или динамическими [2].

Статические библиотеки — это набор объектных файлов, которые компилируются в единую библиотеку. Данная библиотека статически связывается с программой во время линковки, что означает вставку всех функций и данных из библиотеки в исполняемый файл программы. Таким образом, программа может быть запущена где угодно, без необходимости установки дополнительных библиотек.

Динамические библиотеки — это также набор объектных файлов, которые компилируются в единую библиотеку, но при этом библиотека не связывается

статически с программой во время компиляции. Вместо этого она загружается в память во время выполнения программы. Таким образом, программа может использовать функции и данные из библиотеки без необходимости копирования их в единый исполняемый файл программы. Динамические библиотеки обычно используют для экономии памяти и упрощения обновления библиотек.

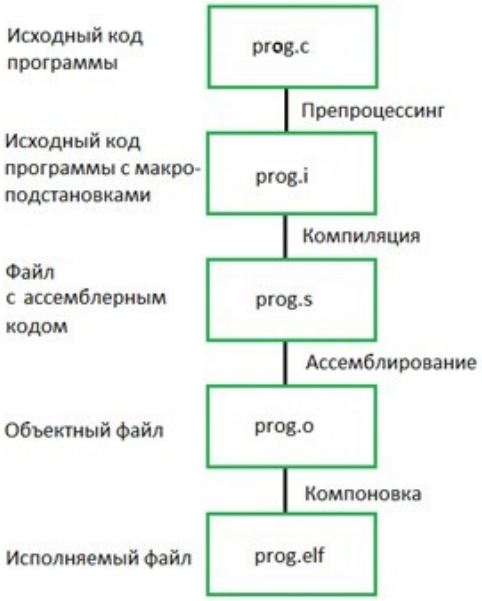

Рис. 1. Этапы компиляции

В целом статические библиотеки способствуют более быстрому выполнению программы, но могут занимать больше места в памяти. Динамические библиотеки обеспечивают более экономичное использование памяти, но могут замедлить выполнение программы из-за необходимости загрузки библиотеки в память во время выполнения.

Объявление и определение. В программировании выделяют два термина при работе с функциями, переменными и т. д. — объявление и определение. Определение связывает имя с реализацией этого имени, т. е. в результате определения переменной компилятор выделит под нее память. Объявление, в свою очередь, — это всего лишь способ сообщить компилятору, что где-то в коде (или, что вполне возможно, в другом файле) будет определение, связанное с этим именем.

В зависимости от того, как объявлена функция или переменная, их подразделяют на локальные и глобальные (по времени их существования и области видимости). Переменные также могут быть инициализированными (т. е. изначально заполненными каким-то значением) и неинициализированными [1].

Анализ процесса ассемблирования программных систем. Ассемблирование программных систем — это процесс преобразования исходного кода программы, написанной на языке ассемблера, в машинный код, который может быть исполнен процессором компьютера. Это важный этап в разработке программных систем, особенно в случае написания системного ПО, которое работает непосредственно с аппаратным обеспечением компьютера. В процессе ассемблирования программа проверяется на наличие ошибок и оптимизируется для более эффективного исполнения на целевой платформе.

Ассемблирование начинается с создания таблицы символов, которая содержит информацию обо всех символах, используемых в программе, таких как метки, макросы, регистры и др. Затем происходит проход по исходному коду, в котором каждая инструкция ассемблера преобразуется в соответствующую машинную инструкцию. При этом происходит также замена символьных ссылок на фактические адреса в памяти, что позволяет правильно связать все части программы. В процессе ассемблирования также происходит разрешение возможных конфликтов между символами, участвующими в программе.

В результате ассемблирования получается объектный файл, содержащий блоки машинного кода, информацию о символах и ссылках, список своих процедур и данных. Объектный файл может быть связан с другими объектными файлами и библиотеками для создания исполняемого файла или динамической библиотеки.

Анализ работы компоновщика. Компоновщик собирает код и данные каждого объектного модуля в окончательную программу [1]. Рассмотрим компоновщик на примере одного из самых популярных, а именно GNU linker (ld). Как сказано в документации к данной программе, ld объединяет ряд объектных и архивных файлов, перемещает их данные и связывает ссылки на символы. Обычно последним шагом в компиляции программы является запуск ld.

Компоновщик ld принимает файлы командного языка компоновщика, написанные в расширенном наборе синтаксиса командного языка редактора ссылок AT&T, чтобы обеспечить явный и полный контроль над процессом компоновки.

Для работы с объектными файлами ld использует библиотеки Binary File Descriptor (BFD) общего назначения. Это позволяет ld читать, комбинировать и записывать объектные файлы в различных форматах, например, COFF или a.out. Библиотека BFD сводит любой формат к стандартному виду и предоставляет для обработки компоновщику, а после обработки линкером переводит обратно в специфичный формат. Таким образом различные форматы могут быть связаны вместе для создания любого доступного типа объектного файла. Этот метод имеет и недостаток, в результате объединения нескольких разных файлов часть специфичной для одного из форматов информации будет потеряна [3].

Анализ библиотеки BFD. В целом BFD представляет собой библиотеку, позволяющую приложениям использовать одни и те же процедуры для работы с объектными файлами независимо от их формата.

Библиотека BFD разделена на две части: внешнюю и внутреннюю (по одной для каждого формата объектных файлов) [4]:

– внешняя часть BFD предоставляет собой высокоуровневый интерфейс. Она отвечает за такие задачи, как открытие и закрытие файлов, вызов внутренних подпрограмм, синтаксический анализ содержимого файла и предоставление интерфейса для управления памятью и различными структурами данных;

– внутренняя часть отвечает за низкоуровневую работу с определенным форматом двоичного файла. Она предоставляет набор функций для чтения и записи двоичных данных, управления памятью и выполнения низкоуровневых операций, таких как, например, управление разделами.

Библиотека BFD работает, представляя общий интерфейс частей объектных файлов для служебных приложений. Объектный файл имеет заголовок с описательной информацией, переменное количество разделов, каждый из которых имеет имя, некоторые атрибуты, блоки данных, таблицу символов и так далее.

BFD-данные преобразуются из абстрактного представления в детали расположения битов/байтов, требуемые целевым процессором и форматом файла. Ключевые задачи BFD включают обработку различий в порядке следования байтов, например, между хостом с прямым порядком байтов и целью с обратным порядком байтов, правильное преобразование между 32-битными и 64-битными данными и детали арифметики адресов, указанные в записях о перемещениях.

Далее компоновщик обращается к объектным и архивным файлам с помощью библиотек BFD. Эти библиотеки позволяют компоновщику использовать одни и те же процедуры для работы с объектными файлами независимо от формата объектных файлов. Другой формат объектного файла можно поддерживать, просто создав новую внутреннюю часть BFD и добавив ее в библиотеку. Однако для экономии оперативной памяти компоновщик и связанные с ним инструменты обычно настраивают на поддержку только подмножества доступных форматов объектных файлов.

Как и в большинстве случаев, BFD — это компромисс между несколькими конфликтующими требованиями. Основным фактором, повлиявшим на разработку BFD, была эффективность: любое время, затраченное на преобразование между форматами, — это время, которое не было бы потрачено, если бы BFD не был задействован. Это частично компенсируется окупаемостью абстракции. Поскольку BFD упрощает интерфейс и внутреннюю часть, можно потратить больше времени и усилий на оптимизацию алгоритмов для повышения скорости [4].

Однако, как уже было упомянуто выше, при использовании BFD для преобразования из более широкого формата в более узкий возможны потери информации. Есть два этапа, где полезная информация может быть потеряна при использовании механизма BFD: во время преобразования и во время вывода.

Потеря информации во время вывода обусловлена несоответствием возможностей выходных форматов, т. е. информация, которая может быть изложена в одном формате, не может быть использована в другом формате. Это происходит, например, когда файлы COFF, содержащие неограниченное количество именованных разделов, ссылаются на файлы, формат которых не имеет такого количества разделов или же разделы безымянные. Но эта проблема решается с помощью описания сопоставления разделов ввода-вывода.

Также информация может быть потеряна во время канонизации. Внутренняя каноническая форма внешних форматов BFD не является исчерпывающей. Существуют структуры во входных форматах, для которых нет прямого внутреннего представления. Это означает, что внутренняя часть BFD не может поддерживать всю возможную полноту данных за счет преобразования внешнего формата во внутренний и обратно во внешний [4].

Анализ процесса линковки программных систем. Как уже стало понятно, существуют разные форматы объектных файлов, используемые линковщиком. Современные UNIX системы используют формат UNIX ELF (executable and linking format). Этот формат содержит следующие разделы [3]:

– .text — машинный код скомпилированной программы;

– .rodata — read-only data, неизменяемая информация, как, например, форматы строк для функции printf;

– .data — в этой секции происходит инициализация глобальных переменных;

– .bss — неинициализированные глобальные переменные. BSS (начало блока данных — block storage start), эта секция обычно пустует в объектных файлах, этакая заглушка;

– .symtab — таблица символов, содержащая информацию о функциях и глобальных переменных, определенных и адресованных в коде программы. Эта таблица не содержит записей о локальных переменных, эта информация содержится на стеке;

– .rel.text — список мест в секции .text, которые необходимо модифицировать, когда линковщик будет компоновать этот объект с другими объектными файлами;

– .rel.data — информация о релокации глобальных переменных, которые объявлены, но не определены в текущем модуле программы;

– .debug — таблица отладочных символов с записями о локальных и глобальных переменных. Эта секция будет присутствовать только если компилятору был передан флаг компиляции с таблицей отладочных символов (-g для gcc);

– .line — отображение номеров строк в исходном C-файле и машинными кодами инструкций. Эта информация необходима для отладки программ;

– .strtab — таблица строк для таблицы символов *.symtab* и секции *.debug.*

В контексте этапа линковки программы библиотека — это набор предварительно скомпилированных объектных файлов, содержащих реализацию функций, которые могут быть использованы в программе.

Как было сказано ранее, выделяют статические и динамические библиотеки. Статическая библиотека представляет собой архив объектных файлов. В таких библиотеках хранятся уже скомпилированные программы, в процессе линковки все необходимые части таких библиотек вносятся в вашу компилируемую программу линковщиком. Происходит это по простому алгоритму: создаются наборы файлов O, U и D, где O — перемещаемые объекты, U — неразрешенные (пока неопределенные) символы, D — определенные символы. Статическая библиотека передается на вход компоновщику, все элементы сканируются, и если в каком-то объектном модуле содержится определение элемента из набора U, то этот модуль включается в набор O, наборы U и D обновляются. Таким образом, если в конце набор U не пустой, линкер выдаст ошибку, в противном случае набор O будет включен в исполняемый файл вместе с исходным объектным и этап линковки будет завершен.

Отметим, что последней стадией в линковке статических библиотек является релокация, которая состоит из двух этапов. Сначала все секции одного типа объединяются вместе, к примеру, все секции .data группируются так, чтобы образовать одну. Затем линковщик указывает текущий адрес памяти для этой сгенерированной секции. Так происходит для каждой секции и символа. После завершения этого шага каждая инструкция и глобальная переменная в программе будет иметь уникальный адрес в момент загрузки.

Следующий этап — релокация символов непосредственно внутри секции, для корректного выполнения инструкций. Здесь используются структуры relocation entries, которые содержат [3]:

– смещение (offset). Для перемещаемых файлов значение смещения — это смещение в байтах от начала секции до получившегося после релокации адреса;

– символ (symbol). Индекс символа в символьной таблице;

– тип (type). Тип релокации (всего стандартом ELF определено 11 типов, но по своей сути они сводятся к абсолютной и относительной релокации).

В целом компоновщик использует различные разделы файла ELF для объединения кода и данных из нескольких входных файлов в единый исполняемый файл или общую библиотеку. Компоновщик разрешает ссылки на символы между входными файлами, генерирует информацию о перемещении и помещает окончательные разделы кода и данных в правильные ячейки памяти в выходном файле. Вот краткий обзор того, как компоновщик использует каждый из разделов [3]:

– .text: компоновщик объединяет все разделы кода из входных файлов в единый исполняемый файл или общую библиотеку. Он также выполняет любые необходимые корректировки кода для разрешения ссылок на символы между входными файлами;

– data: компоновщик объединяет все инициализированные разделы данных из входных файлов в единый раздел данных в выходном файле. Выполняются

любые необходимые корректировки данных для разрешения ссылок на символы между входными файлами;

– bss: компоновщик резервирует место для раздела неинициализированных данных в выходном файле;

– rodata: компоновщик объединяет все доступные только для чтения разделы данных из входных файлов в один доступный только для чтения раздел данных в выходном файле;

– .symtab: компоновщик объединяет все таблицы символов из входных файлов в единую таблицу символов в выходном файле. Проводятся корректировки таблицы, необходимые для разрешения ссылок на символы между входными файлами;

– .rela.text и .rela.data: компоновщик генерирует информацию о перемещении для разделов кода и данных соответственно. Информация о перемещении используется операционной системой для настройки адресов символов в выходном файле таким образом, чтобы они указывали на правильные ячейки памяти во время выполнения.

Таким образом, компоновщик использует различные разделы в формате файла ELF для объединения кода и данных из нескольких входных файлов в единую исполняемую или совместно используемую библиотеку, разрешения ссылок на символы между входными файлами и генерации информации о перемещении для настройки адресов символов в выходном файле.

Анализ статической и динамической линковки. Статическая библиотека копируется в исполняемый файл, соответственно, занимает дополнительное место. Если написанная библиотека уникальна и не используется в других программах, то это отличный вариант, ведь программа получается самодостаточной, однако в случае часто используемой библиотеки такой подход становится неэффективным. Например, знакомая всем функция printf используется повсеместно многими процессами системы, каждый раз включать ее отдельно означает потратить много дискового пространства машины в пустую, кроме того, при обновлении функции printf придется пересобрать все эти программы заново. В таких случаях используют динамические библиотеки.

При линковке с динамической библиотекой создается исполняемый файл, в котором кроме собственных объектных файлов содержатся лишь информация для будущих релокаций и символьная таблица. Сам выходной файл не является автономной единицей, поскольку при его запуске (загрузке в память) также запускается динамический линкер, имя которого хранится в специальной .interp секции ELF-файла. Этот линкер осуществляет релокацию объектов библиотеки в память и таким образом связывает исходный ELF с динамической библиотекой [3].

Преимуществами такого вида сборки являются: заметно меньший объем, занимаемый на дисковом пространстве машины, удобство в последующей под-

держке программы (исправления в функции нужно вносить лишь в исходной библиотеке), гибкость в работе (динамический линкер может подгрузить необходимые объекты прямо во время работы программы, если это необходимо), загруженные в память объекты могут быть разделены между несколькими процессами.

Сравнение статической и динамической линковки. Для того, чтобы в итоге можно было приступить к линковке нужно создать объектный файл:

```
gcc -c test_dynamic.c -o test_dynamic.o
```

Выполним линковку (по умолчанию gcc осуществляет динамическую линковку):

```
gcc -o test_dynamic test_dynamic.o
```

Для определения времени, затраченного на линковку, используем утилиту time (рис. 2).

```
sarwx@IU4:~/test/dynamic$ time gcc -o test_dynamic test_dynamic.o

real    0m0,175s
user    0m0,023s
sys     0m0,000s
```

Рис. 2. Определение времени динамической линковки

Определим размер получившегося исполняемого файла в байтах (рис. 3).

```
sarwx@IU4:~/test/dynamic$ stat test_dynamic
  Файл: test_dynamic
  Размер: 16096         Блоков: 32         Блок В/В: 4096   обычный файл
Устройство: 803h/2051d  Инода: 1439030     Ссылки: 1
Доступ: (0775/-rwxrwxr-x)  Uid: ( 1000/   sarwx)   Gid: ( 1000/   sarwx)
Доступ:        2023-04-30 15:32:04.815457531 +0300
Модифицирован: 2023-04-30 15:32:04.815457531 +0300
Изменён:       2023-04-30 15:32:04.815457531 +0300
Создан:        2023-04-30 15:32:04.735456074 +0300
```

Рис. 3. Определение размера исполняемого файла

Для подтверждения того факта, что исполняемый файл действительно получился с помощью динамической линковки, воспользуемся утилитой file (рис. 4).

```
sarwx@IU4:~/test/dynamic$ file test_dynamic
test_dynamic: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=5ba96606fdb4077ff8704777b8122c621a90b2fd, for GNU/Linux 3.2.0, not stripped
```

Рис. 4. Проверка исполняемого файла

Можно сделать промежуточный вывод, что при использовании динамической линковки время линковки равно 0,175 с. Объем памяти, которую занимает исполняемый файл, составляет 16 096 байт.

Итоговый листинг команд для создания исполняемого файла с применением динамической линковки приведен на рис. 5.

```
sarwx@IU4:~/test/dynamic$ ls
test_dynamic.c  test.txt
sarwx@IU4:~/test/dynamic$ gcc -c test_dynamic.c -o test_dynamic.o
sarwx@IU4:~/test/dynamic$ ls
test_dynamic.c  test_dynamic.o  test.txt
sarwx@IU4:~/test/dynamic$ time gcc -o test_dynamic test_dynamic.o

real    0m0,175s
user    0m0,023s
sys     0m0,000s
sarwx@IU4:~/test/dynamic$ stat test_dynamic
  Файл: test_dynamic
  Размер: 16096         Блоков: 32         Блок В/В: 4096   обычный файл
Устройство: 803h/2051d  Инода: 1439030     Ссылки: 1
Доступ: (0775/-rwxrwxr-x)  Uid: ( 1000/   sarwx)   Gid: ( 1000/   sarwx)
Доступ:        2023-04-30 15:32:04.815457531 +0300
Модифицирован: 2023-04-30 15:32:04.815457531 +0300
Изменён:       2023-04-30 15:32:04.815457531 +0300
Создан:        2023-04-30 15:32:04.735456074 +0300
sarwx@IU4:~/test/dynamic$ file test_dynamic
test_dynamic: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=5ba96606fdb4077ff8704777b8122c621a90b2fd, for GNU/Linux 3.2.0, not stripped
```

Рис. 5. Итоговый листинг для создания исполняемого файла

Теперь выполним статическую линковку. Для этого создадим объектный файл:

```
gcc -I . -c test_static.c
```

Попробуем воспользоваться статической линковкой:

```
gcc -static -o test_static test_static.o /usr/lib/
                                          x86_64-linux-gnu/libc.a
```

Для определения времени, затраченного на линковку, используем утилиту time (рис. 6).

```
sarwx@IU4:~/test$ time gcc -static -o test_static  test_static.o /usr/lib/x86_64-linux-gnu/libc.a

real    0m0,355s
user    0m0,028s
sys     0m0,013s
```

Рис. 6. Определение времени статической линковки

Определим размер получившегося исполняемого файла в байтах (рис. 7).

```
sarwx@IU4:~/test$ stat test_static
  Файл: test_static
  Размер: 900504        Блоков: 1760       Блок В/В: 4096   обычный файл
Устройство: 803h/2051d  Инода: 1438987     Ссылки: 1
Доступ: (0775/-rwxrwxr-x)  Uid: ( 1000/   sarwx)   Gid: ( 1000/   sarwx)
Доступ:        2023-04-30 15:40:30.397265345 +0300
Модифицирован: 2023-04-30 15:39:52.841822838 +0300
Изменён:       2023-04-30 15:39:52.841822838 +0300
Создан:        2023-04-30 15:39:52.713811137 +0300
```

Рис. 7. Определение размера исполняемого файла

Для подтверждения того факта, что исполняемый файл действительно получился с помощью статической линковки, так же, как и предыдущем примере, воспользуемся утилитой file (рис. 8).

Можно сделать промежуточный вывод, что при использовании статической линковки время линковки равно 0,355 с. Объем памяти, которую занимает исполняемый файл, составляет 900 504 байта.

```
sarwx@IU4:~/test$ file test_static
test_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked,
BuildID[sha1]=ab307660001e3da3938b549d558ee0985e2c3426, for GNU/Linux 3.2.0, not stripped
```

Рис. 8. Проверка исполняемого файла

Листинг команд для создания исполняемого файла с применением статической линковки приведен на рис. 9.

```
sarwx@IU4:~/test$ ls
dynamic  test_static.c  test.txt
sarwx@IU4:~/test$ gcc -I . -c test_static.c
sarwx@IU4:~/test$ ls
dynamic  test_static.c  test_static.o  test.txt
sarwx@IU4:~/test$ time gcc -static -o test_static  test_static.o /usr/lib/x86_64-linux-gnu/libc.a

real    0m0,355s
user    0m0,028s
sys     0m0,013s
sarwx@IU4:~/test$ ls
dynamic  test_static  test_static.c  test_static.o  test.txt
sarwx@IU4:~/test$ ./test_static
Hello world!hello
sarwx@IU4:~/test$ stat test_static
  Файл: test_static
  Размер: 900504        Блоков: 1760       Блок В/В: 4096   обычный файл
Устройство: 803h/2051d  Инода: 1438987     Ссылки: 1
Доступ: (0775/-rwxrwxr-x)  Uid: ( 1000/   sarwx)   Gid: ( 1000/   sarwx)
Доступ:        2023-04-30 15:40:30.397265345 +0300
Модифицирован: 2023-04-30 15:39:52.841822838 +0300
Изменён:       2023-04-30 15:39:52.841822838 +0300
Создан:        2023-04-30 15:39:52.713811137 +0300
sarwx@IU4:~/test$ file test_static
test_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked,
BuildID[sha1]=ab307660001e3da3938b549d558ee0985e2c3426, for GNU/Linux 3.2.0, not stripped
```

Рис. 9. Итоговый листинг для создания исполняемого файла

Результаты. Таким образом, на практике удалось доказать, что итоговый исполняемый файл после статической линковки занимает в разы больше места, чем при динамической линковке. Это связано с тем, что во втором случае все зависимости присоединяются по ходу выполнения программы из динамических библиотек, когда при статической линковке последние заранее прикреплены к исполняемому файлу. Время, затраченное на линковку, в случае динамической линковки меньше, чем при статической.

На основании проделанной работы разработчикам предлагается уделять этапу линковки большее внимание. Хотя и сам по себе этот этап достаточно прост и на данный момент его трудно оптимизировать, разработчикам предлагается использовать два разных способа линковки. Правильный выбор одного из этих способов может оказать существенное влияние на производительность и дальнейшую эффективность распространяемого программного обеспечения.

Для осуществления этого выбора мы предлагаем на этапе разработки ПО следовать простому алгоритму:

1) проанализировать уникальность используемых функций и библиотек;

2) оценить целевое аппаратное обеспечение и условия функционирования ПО;

3) оценить надобность дальнейшей поддержки используемых библиотек;

4) на основании полученных данных сделать выбор в пользу статической линковки, если используемые функции уникальны и не требуют поддержки в дальнейшем, а программа будет функционировать на слабых устройствах вне операционной системы. Во всех остальных случаях выбор следует делать в пользу динамической. При этом сам разработчик может практически сравнить исполняемые файлы, полученные различными способами линковки с помощью листинга, представленного на рис. 5 и 9, и выбрать нужный способ для реализации его проекта.

Заключение. В ходе работы были проанализированы основные этапы линковки, которые необходимы для связывания различных модулей программы в единое целое. В процессе исследования было выявлено, что линковка включает в себя такие этапы, как объединение объектных файлов, разрешение символов, распределение памяти и создание исполняемого файла.

Было установлено, что важность линковки заключается в том, что она позволяет создавать готовые к выполнению программы, которые могут быть запущены на компьютере пользователя. Она также позволяет использовать библиотеки, что позволяет сократить время разработки и улучшить качество кода.

Помимо этого было проведено сравнение статической и динамической линковки с выявлением времени процесса и занимаемой памятью исполняемым файлом. По результатам анализа динамическая линковка показала свои преимущества, она слинковалась быстрее примерно в 4 раза, а объектный файл занимает меньше памяти более чем в 50 раз. Однако благодаря изученным теоретическим данным, можно сделать вывод о том, что статические библиотеки будут выигрышными в случае использования каких-то уникальных библиотек, в примере мы использовали библиотеку libc.a, а она используется программами повсеместно.

Также была разработана методика линковки программных систем, с помощью которой любой разработчик ПО сможет сделать правильный выбор использования библиотек во время линковки при создании своего продукта. Поэтому при разработке любого проекта необходимо анализировать использованные библиотеки и выбирать нужный метод линковки.

Выводы, полученные в ходе исследования, могут быть полезными для программистов, которые занимаются разработкой программного обеспечения, поскольку понимание процесса линковки помогает создавать более эффективный и оптимизированный код, а также начинающим карьеру в области программирования специалистам.

Литература

[1] Брайант Р.Э., О'Халларон Д.Р. *Компьютерные системы: архитектура и программирование.* Москва, ДМК Пресс, 2022, 994 с.

[2] Ахо А., Лам М.С., Сети Р., Ульман Д. *Компиляторы: принципы, технологии и инструменты*. Москва, Диалектика, 2020, 1184 с.

[3] Taylor I.L., Chamberlain S. *The GNU linker. GNU ld. version 2*. GNU ORG, 1994, 96 p.
[4] Chamberlain S. *LIB BFD, the Binary File Descriptor Library. BFD version < 3.0*. GNU ORG, 1991.
[5] Керниган Б., Ритчи Д. *Язык программирования Си*. Москва, Вильямс, 2019, 288 с.
[6] Демин А.А., Карпунин А.А., Ганев Ю.М. Методы верификации и валидации сложных программных систем. *Программные продукты и системы*, 2014, № 4, с. 229–233. http://doi.org/10.15827/0236-235x.108.229-233
[7] Карпунин А.А., Ганев Ю.М., Чернов М.М. Методы обеспечения качества при проектировании сложных программных систем. *Надежность и качество сложных систем*, 2015, № 2 (10), с. 78–84.
[8] Власов А.И., Карпунин А.А., Курышев Р.Э. Визуальное моделирование smart-технологий проектного управления. *Надежность и качество. Тр. междунар. симпозиума: сб. докл.* Пенза, ПГУ, 2020, т. 1, с. 64–70.

Луцук Ренат Александрович — студент кафедры «Проектирование и технология производства электронной аппаратуры», МГТУ им. Н.Э. Баумана, Москва, Российская Федерация.

Мельников Дмитрий Вячеславович — студент кафедры «Проектирование и технология производства электронной аппаратуры», МГТУ им. Н.Э. Баумана, Москва, Российская Федерация.

Новиков Тимофей Александрович — студент кафедры «Проектирование и технология производства электронной аппаратуры», МГТУ им. Н.Э. Баумана, Москва, Российская Федерация.

Научный руководитель — Власов Андрей Игоревич, кандидат технических наук, доцент кафедры «Проектирование и технология производства электронной аппаратуры», МГТУ им. Н.Э. Баумана, Москва, Российская Федерация.

Ссылку на эту статью просим оформлять следующим образом:
Луцук Р.А., Мельников Д.В., Новиков Т.А. Методика линковки программных систем с использованием статических и динамических библиотек. *Политехнический молодежный журнал*, 2023, № 07 (84). http://dx.doi.org/10.18698/2541-8009-2023-7-921

SOFTWARE SYSTEM LINKAGE USING THE STATIC AND DYNAMIC LIBRARIES

R.A. Lutsuk lra20u222@student.bmstu.ru
D.V. Melnikov melnikovdv@student.bmstu.ru
T.A. Novikov nta20u673@student.bmstu.ru

Bauman Moscow State Technical University, Moscow, Russian Federation

Abstract

The paper considers the final stage in compiling a program, namely the linkage stage, i. e. linking the already compiled code fragments and the code of all the other libraries into a single file. Mechanism of the BFD (Binary File Descriptor) library operation and features of its application are described. Static and dynamic libraries were compared. Linker operation and its interaction with various object files were analyzed using the GNU linker, as the example. Besides, advantages and disadvantages of the static and dynamic libraries were considered in terms of such parameters, as the occupied memory of the executable file and the linkage period.

Keywords

Programming, software, design, compilation, linkage, linker, static and dynamic libraries, project assembly

Received 05.05.2023

References

[1] Bryant R.E., O'Hallaron D.R. *Computer Systems: A Programmer's Perspective*. Pearson India Education Services Pvt. ltd., 2016, 1043 p. (Russ. Ed.: Brayant R.E., O'Khallaron D.R. *Komp'yuternye sistemy: arkhitektura i programmirovanie*. Moscow, DMK Press Publ., 2022, 994 p.).

[2] Aho A.V., Lam M.S., Sethi R., Ullman J.D. *Compilers Principles, Techniques, and Tools*. Pearson Education Inc., 2007, 1038 p. (Russ. Ed.: Akho A., Lam M.S., Seti R., Ul'man D. *Kompilyatory: printsipy, tekhnologii i instrumenty*. Moscow, Dialektika Publ., 2020, 1184 p.).

[3] Taylor I.L., Chamberlain S. *The GNU linker. GNU ld. version 2*. GNU ORG, 1994, 96 p.

[4] Chamberlain S. *LIB BFD, the Binary File Descriptor Library. BFD version < 3.0*. GNU ORG, 1991.

[5] Kernighan B., Ritchie D. *The C Programming Language*. Prentice Hall, 1988, 217 p. (Russ. Ed.: Kernigan B., Ritchi D. *Yazyk programmirovaniya Si*. Moscow, Vil'yams Publ., 2019, 288 p.).

[6] Demin A.A., Karpunin A.A., Ganev Yu.M. Verification and validation methods for complex software systems. *Programmnye produkty i sistemy*, 2014, no. 4, pp. 229–233. (In Russ.). http://doi.org/10.15827/0236-235x.108.229-233

[7] Karpunin A.A., Ganev Yu.M., Chernov M.M. Quality assurance methods in the design of complex software systems. *Reliability & quality of complex systems*, 2015, no. 2 (10), pp. 78–84. (In Russ.).

[8] Vlasov A.I., Karpunin A.A., Kuryshev R.E. Visual modeling of smart-technologies of project management. *Nadezhnost' i kachestvo. Tr. mezhdunar. simpoziuma: sb. dokl.* [Reliability and quality. Proceedings of the international symposium: collection of papers]. Penza, PGU Publ., 2020, vol. 1, pp. 64–70. (In Russ.).

Lutsuk R.A. — Student, Department of Design and Technology of Production of Electronic Equipment, Bauman Moscow State Technical University, Moscow, Russian Federation.

Melnikov D.V. — Student, Department of Design and Technology of Production of Electronic Equipment, Bauman Moscow State Technical University, Moscow, Russian Federation.

Novikov T.A. — Student, Department of Design and Technology of Production of Electronic Equipment, Bauman Moscow State Technical University, Moscow, Russian Federation.

Please cite this article in English as:
Lutsuk R.A., Melnikov D.V., Novikov T.A. Software system linkage using the static and dynamic libraries. *Politekhnicheskiy molodezhnyy zhurnal*, 2023, no. 07 (84). (In Russ.). http://dx.doi.org/10.18698/2541-8009-2023-7-921