Ян Малаховски

Конспект лекций по курсу «Операционные системы»

version 3
compiled 23 мая 2011 г.

Оглавление

Введение **2**

I Первый семестр **4**

II Второй семестр **6**

1 Юристы и компьютеры **7**
1.1 Юридические ужасы . . . 7
1.2 Лицензии, GNU, Creative Commons и другие . . . 9
1.3 Почему свободности недостаточно . . . 11

2 Приложения и система: низкий уровень **13**
2.1 ABI и системные вызовы . . . 13
2.1.1 ABI . . . 13
2.1.2 Системные вызовы . . . 14
2.2 Компиляция, линковка, динамическая загрузка и связанные с этим вещи . . . 17
2.2.1 Объектные файлы . . . 17
2.2.2 Линковка и динамическая загрузка . . . 19
2.2.3 ELF . . . 27
2.2.4 Секции и сегменты . . . 29
2.2.5 Линковка . . . 29
2.2.6 Динамический загрузчик . . . 30
2.3 Практика . . . 30
2.4 Комментарии . . . 31
2.5 Краткое содержание . . . 31

Введение

Документ, который вы читаете, представляет собой инновационный конспект по курсу «Операционные системы». Инновационность заключается в постоянном использовании буквы «ё», где надо и где нет (соответственно, в версии без буквы «ё» эта инновационность отсутствует всюду кроме этого введения).

Всё, что не касается технической стороны предметов, затрагиваемых в данном документе, является моей личной точкой зрения, которая не может считаться коль сколь-нибудь авторитетной в вопросах, в которых я не считаю себя экспертом (например, авторское право). Тем не менее, любые конструктивные замечания и исправления принимаются с радостью.

Соглашения

Определения и теоремы выделяются классически, используется сквозная нумерация внутри глав. Словосочетания, важности которых для данного текста не хватает, чтобы носить гордое название «Определение», выделены *курсивом*, точно также выделяются места с логическим ударением.

Определение 0.1. *Научная работа — работа в которой есть хотя бы одно доказательство.*

Теорема 0.2. *Этот текст — научная работа.*

Доказательство. Тут есть доказательство. □

Все приводимые примеры кода по возможности компилируются и оттестированы. В заголовках листингов указывается имя файла, в котором его нужно разместить, а в комментариях в конце листинга иногда указываются команды, которые необходимо ввести в оболочку, чтобы этот файл скомпилировать (слинковать, запустить, ...). Листинги без заголовков представляют собой, или псевдокод, или вывод каких-то программ, или примеры, которые невозможно запустить по каким-то причинам.

По отношению к структурному элементу многих языков программирования, обычно называемому «функцией», (из религиозных соображений) применяется термин «процедура» в не зависимости от того, имеется ли у этого элемента возвращаемое значение.

Все факты из данного документа настоятельно рекомендуется сверять с соответствующими мануалами, а все исходники изучать, компилировать и запускать, поскольку они могут содержать как случайные, так и не очень, ошибки.

Состав

В каждый момент времени этот документ не окончен, находится в разработке и в бета-версии. Секции и даже целые главы могут быть пропущены из-за моей лени, недостатка времени или слишком низкого приоритета (например, излагается очень простой материал, съедобную информацию по которому легко найти в сети).

Курс рассчитан на два семестра (а может быть даже и на три). В первом семестре проводится общее знакомство с системами типа UNIX-like без каких-либо серьёзных подробностей. Примерно половина

второго семестра посвящена трешу POSIX API, а вторая половина — каким-то более-менее общим (а потому хоть немного интересным) вещам. Третий семестр как бы рассчитан на advanced темы и мощных студентов и, если когда-нибудь и будет существовать, то, видимо, в качестве факультатива (семинаров, или чего-то такого).

В принципе, первый семестр — абсолютный треш и всё что в нём содержится можно самостоятельно изучить за неделю или две, ещё есть какие-то лабораторные работы, но они тупые, студентам лень их делать, а мне проверять. В каждую конкретную «интересную» тему второго семестра можно закопаться очень глубоко, однако это делать опасно, ибо можно обратно уже не вылезти. Про третий семестр ничего не знаю.

Часть I

Первый семестр

Тут когда-нибудь будут лекции по первому семестру. Или не будут.

Часть II

Второй семестр

Глава 1

Юристы и компьютеры

О юридических ужасах, подводящих к необходимости существования лицензий в этом страшном мире, свободном и открытом программном обеспечении, а также почему оно нужно, но не достаточно.

1.1 Юридические ужасы

Computer science и право различаются уже хотя бы в том, что законодателями, и в этой, и в той стране, в основном являются юристы, и, если без «операционных систем» в какой-то области можно обойтись, то без минимальных юридических знаний можно случайно совершить какое-нибудь серьёзное правонарушение и получить много проблем. Не менее прискорбно то, что законодатели обычно не знают теоретических основ computer science (теории вычислительной сложности, например), а потому содержание многих юридических документов, имеющих отношение к этой области знания, может вызывать недоумение. Это уже не говоря о том, что почти все юридические тексты написаны по таким правилам и таким языком, что очень часто их просто невозможно понять и адекватно толковать, не посвятив этому жизни.

В этом разделе я постараюсь изложить своё видение необходимых человеку из computer science юридических терминов. Все права на упоминаемые в этом разделе торговые марки принадлежат их законным правообладателям.

Определение 1.1. *Интеллектуальная собственность — множество, включающее в себя* авторские и смежные права, торговые марки, патенты, *а также другие, не менее страшные, но менее известные юридические термины.*

Определение 1.2. *Торговая марка — некоторая строка символов, право использования которой в каких-то целях охраняется на государственном уровне. Является ли логотип торговой маркой мне не ясно.*

Например, строка «Coca-Cola» является торговой маркой. Считается, что торговые марки появились в средние века с той целью, чтобы один ремесленник мог выделить продукт своего производства из таких же продуктов, производимых другими ремесленниками, но так, что право на использование имени, которым данный ремесленник обозначал свои продукты, находилось под охраной государства. Тогда другого ремесленника, который делал бы неавторизованные данным ремесленником подделки, можно было бы как-то наказать. Таким образом, торговые марки были придуманы для защиты потребителей от потенциально некачественного товара.

Однако в современном мире торговые марки используются как раз для того, чтобы обманывать потребителей. Например, товарная марка «Coca-Cola» когда-то обозначала напиток, содержащий вещества из растения «соса», но скоро в США лавочку с использованием этих веществ прикрыли (объявив их наркотическими), однако потребителей до сих пор обманывают названием данной торговой марки напитка, в котором экстракта этого растения уже нет.

Не менее выгодным бизнесом оказалась продажа торговых марок «в аренду», при такой схеме, владельцу раскрученной торговой марки вообще можно ничего не делать, а только взымать «арендную плату» за использование строки символов. Например, «McDonald's» в разных странах представлен совершенно разными компаниями, «арендующими» эту торговую марку у оригинальной компании в США. Для защиты потребителей от такого обмана в некоторых странах были приняты законы, запрещающие передавать права на пользование торговой маркой без передачи технологий, используемых в оригинальном производстве. Это лучше чем совсем ничего, однако это не мешает «арендующему» производителю в другом государстве не соблюдать технических процессов, использовать другое сырьё и так далее.

Торговые марки обозначаются знаками «™» (trade mark) и «®» (registered trade mark), тонкостей разницы между этими двумя понятиями я не знаю.

Определение 1.3. *Авторство — констатация того факта, что кто-то сделал что-то.*

Например, Бетховен является автором своей девятой симфонии, и его авторство от самой симфонии оторвать никак нельзя, даже если бы Бетховен, в своё время (но по современным законам), и хотел бы продать авторство на эту симфонию, он бы этого сделать не смог.

Определение 1.4. *Научный приоритет — тоже самое, что и авторство, но для научных открытий.*

Пифагор является автором нескольких теорем, подобно тому как Бетховен является автором не только одной симфонии. Однако, когда речь идёт о научных открытиях, это почему-то называют не авторством, а научным приоритетом.

Определение 1.5. *Авторское право — права в отношении использования произведения, которыми наделяется его автор.*

Авторские права, в отличии от авторства, можно кому-то передать.

Определение 1.6. *Общественное достояние — то, куда попадают все произведения, авторские права на которые истекли.*

Вернёмся к Бетховену, почему-то живущему по современным законам. Пусть Бетховен сегодня сочинил свою девятую симфонию. Этот факт делает Бетховена автором симфонии, и, кроме того, наделяет его в отношении неё авторскими правами. Например, он, а после его смерти и его потомки в течении какого-то срока (в России это что-то типа семидесяти лет, если я не ошибаюсь), могут требовать денег за любое её использование. После истечения этого срока симфония перестаёт охраняться авторскими правами и переходит в общественное достояние. С произведениями в общественном достоянии кто угодно может делать что угодно.

Законодательство некоторых стран (но не России) позволяет автору самостоятельно отказаться от своих авторских прав и передать своё произведение в общественное достояние.

Авторские права, по большому счёту, означают полный контроль над тем, кем и как используется созданное произведение в течении строка до попадания произведения в общественное достояние. Например, Бетховен мог бы разрешить записывать и продавать диски с его симфонией какой-то одной компании, при этом запретив ей распространять свою симфонию, например, в Африке. Кроме того, Бетховен имел бы, например, право опубликовать симфонию под своим именем, под псевдонимом или вообще анонимно. Когда речь идёт о правах на копирование и воспроизведение произведения — это называется *имущественным авторским правом*, или *копирайтом* (copyright). В других случаях — *неимущественным авторским правом*.

Определение 1.7. *Смежные права — права, которыми наделяется исполнитель произведения, если исполнение требует техники, оборудования, таланта, мастерства, и тому подобного.*

Например, смежными называются права, которыми обладает оркестр, исполнивший девятую симфонию Бетховена, и звукозаписывающая компания, выпустившая этот концерт на диске.

Определение 1.8. *Патент — исключительное право на использование результатов изобретения.*

Патенты были изобретены в качестве метода борьбы с секретами производства, уходящими в могилу вместе с мастерами, их использующими. Идея заключалась в том, чтобы дать ремесленникам способ, при помощи которого можно было бы раскрыть миру секреты своего производства, но так, чтобы без разрешения автора или его потомков в течении какого-то срока (что-то около двадцати лет, если я не ошибаюсь) никто не смог бы эти секреты использовать.

Проблема заключается в том, что если секрет действительно очень важный, то выгоднее его держать при себе и вообще не патентовать, зато запатентовав что-нибудь, что используют или будут использовать все, можно ничего не делать и красиво жить. Таким образом в современном мире патенты превратились в средство шантажа и метод монополизации целых отраслей производств. Например, в индустрии производства ПО монополия длиной в двадцать лет на какой-то тип софтварных продуктов практически равносильна монополии навсегда.

Основная логика, прослеживающаяся в различиях между авторскими правами и патентным правом, заключается в том, что патентное право предоставляет исключительные права первооткрывателям каких-то фактов или конструкций (не смотря на то, что, потенциально, до любого открытия могут дойти два и более независимых учёных, исключительное право на использование предоставляется первому), а авторские права защищают произведения, которые (почти достоверно) не могут быть повторены двумя независимыми авторами (не смотря на утверждение про миллион обезьян за печатными машинками, маловероятно, что одну и туже «Войну и мир», почти одновременно напишут два независимых писателя).

Тем не менее, существуют области знаний которые вообще не поддаются охране авторскими правами и патентами, например, математические результаты (хотя сам текст, содержащий математический результат может быть защищён авторскими правами в качестве литературного произведения). С другой стороны, во многих странах программы можно защитить авторскими правами (тем самым исходники приравнены к литературным произведениями), а в некоторых странах можно даже запатентовать программные алгоритмы.

Однако заметим, что теория вычислительной сложности утверждает, что любой алгоритм представим в виде программы для универсальной машины Тьюринга, а любую программу на любом языке программирования можно преобразовать в эквивалентную программу для машины Тьюринга. Любая программа для машины Тьюринга представима в виде строки символов, которую можно интерпретировать как запись, например, натурального числа (потенциально очень большого, но всё же). Таким образом, задача поиска программы, обладающей требуемыми свойствами, сводится к поиску какого-то натурального числа (а на самом деле даже ряда чисел). Возникает вопрос: «Можно ли запатентовать натуральные числа и не является ли программа, а заодно и алгоритм, математическим результатом?»

В общем, не всё так гладко, как хотелось бы. Почти всё, что изложено в этом разделе было почёрпнуто из:

- `http://gorod.tomsk.ru/index-1248681368.php`;
- `http://www.groklaw.net/article.php?story=20091111151305785`;
- Гражданский Кодекс РФ, часть IV.

1.2 Лицензии, GNU, Creative Commons и другие

Лицо, обладающее копирайтом на какой-то объект, может этим своим копирайтом с кем-нибудь поделиться. Например, автор программы может предоставить кому-то конкретно (или вообще всем) право безвозмездно ей пользоваться. Если предоставляется право использования и модификации исходных кодов, то автор может потребовать, например, чтобы ему был предоставлен доступ к результатам модификаций.

Определение 1.9. *Лицензия — правила, в соответствии с которыми третье лицо может использовать объект, охраняемый авторскими правами.*

Идея тут такая: по умолчанию никто кроме автора не может пользоваться его произведением, а при помощи лицензии автор устанавливает правила, выполняя которые третьи лица могут использовать его произведение. Лицо, нарушающее условие лицензии, по которой распространяется объект, теряет заодно и все права, предоставляемые ему этой лицензией.

Если вы вдруг помните хоть что-то из истории UNIX-подобных операционных систем, то вас не удивит тот факт, что в 1983 году Ричард Столлман уходит из MIT для того, чтобы заняться разработкой своей свободной операционной системы GNU, поскольку по трудовому договору авторские права на всё, сделанное за время работы в MIT, принадлежат MIT. Основными результатами, полученными в ходе разработки GNU, являются лицензия *GNU GPL* (GNU General Public License), набор компиляторов (самый известный из которых — *GCC*) и набор стандартных юзерспейс-утилит, соответствующих стандарту POSIX. И компиляторы, и утилиты распространяются по лицензии GPL.

Идеология, продвигаемая Столлманом и *FSF* (Free Software Foundation — организация, «выросшая» из GNU, сейчас GNU является одним из проектов FSF), основана на понятии *свободного программного обеспечения*, содержащего четыре основных свободы.

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements (and modified versions in general) to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

В английском языке с термином «свободный» обнаруживаются неожиданные проблемы, которые можно выразить известной фразой «Free as in Freedom or free as in beer» (мой вольный перевод: «Свободный как Свобода, или свободный как халявное пиво»). В поисках лучшего синонима Эрик Реймонд придумал термин «*открытое программное обеспечение*», который сегодня используется намного чаще. Однако в за этот термин яро ухватились крупные корпорации, распространяющие плоды своей деятельности в виде общедоступных исходных кодов, но под лицензиями, допускающими только некоторые из четырёх вышеназванных свобод. Другими словами, по факту, сегодня любое свободное ПО является открытым, но не любое открытое — свободным. В том числе и поэтому, Столлман регулярно выступает на разных конференциях и пишет статьи о том, что не стоит путать эти два термина.

Лицензии, предоставляющие четыре вышеупомянутых свободы всем пользователям, в противоположность копирайту, принято называть *копилефтом* (copyleft).

Первая версия GNU General Public License создавалась в качестве единой лицензии для всего свободного ПО и во многом объединяла общие части лицензий, использовавшихся при распространении программ того времени. Однако кроме даров пользователям ПО, состоящих из четырёх свобод, GPL требует, чтобы всё, использующее код, распространяемый под GPL, также распространялось под GPL. Именно поэтому некоторые важные в крупных корпорациях лица в своё время называли GPL «вирусом» индустрии разработки ПО.

Поскольку любая лицензия является юридическим текстом, а, как известно, юридические тексты очень часто содержат не достаточно формально определённые понятия, то как только кто-то находит дыру в возможности трактовки того или иного куска существующей лицензии, так сразу в эту дыру начинают ломиться все кому не лень. Потому FSF иногда выпускает новые версии своих лицензий, поддерживая их в актуальном состоянии, и закрывая «юридические баги».

Вместе со второй версией GPL на свет появилась лицензия *GNU LGPL* (GNU Library General Public License, позже переименованная в GNU Lesser General Public License), разрешающая использовать код программы без модификации, вместе с кодом, не распространяющимся под LGPL, однако в случае модификации LGPL-кода результаты также должны распространяться под LGPL. Как ни странно, но первой версии LGPL сразу была назначена вторая версия, дабы подчеркнуть её связь с GPLv2.

Третья версия GPL (а заодно и LGPL) была опубликована в 2007-ом году и стала запрещать *тивоизацию* (tivoization — аппаратное ограничение модификации прошивок устройств, получаемых с использованием свободного ПО, например, ограничение на замену прошивок в мобильных телефонах, основанных на ядре Linux), использование *DRM* (Digital Rights Management — программно-аппаратные средства защиты авторских прав на произведения в цифровом формате), передачу прав на использование патентов только некоторым пользователям кода (чтобы одних пользователей любить, а других — судить) и так далее. GPLv3 вызывала много споров о необходимости столь строгих ограничений ещё до публикации окончательной версии, и некоторые проекты (например, ядро Linux) менять лицензию не стали. Также в 2007-ом году появилась *GNU AGPL* (GNU Affero General Public License), требующая доступ к исходным кодам ПО, с которыми программа, распространяемая под AGPL, общается по сети.

Однако лицензий GNU на все случаи жизни не хватает. Например, в мире операционных систем семейства BSD используются всевозможные модификации оригинальной *BSD License* (Berkley Software Distribution), фактически разрешающей любое использование оригинального кода, кроме перелицензирования. Также достаточно часто можно встретить *MIT License* (она же *X11 License*), требующую распространения текста самой лицензии со всеми модифицированными версиями кода. Многие литературные произведения, а также всякие другие произведения искусства, распространяются по лицензиям из семейства *Creative Commons*. В общем, всех не перечислить. Сводную табличку с разными лицензиями и их свойствами можно наблюдать в Википедии по адресу `http://en.wikipedia.org/wiki/Comparison_of_free_software_licenses`. Там же можно обнаружить, что некоторые не-GNU лицензии признаются FSF «свободными», а некоторые — признаются *OSI* (Open Source Initiative — ещё одна организация, эксплуатирующая термин «открытое ПО», подобно тому как, FSF эксплуатирует термин «свободное ПО») «открытыми», при чём пересечение этих множеств не совпадает ни с одним из них.

Но, конечно, никто не может запретить автору распространять один и тот же код под несколькими лицензиями, например, часто можно встретить комбинации из GPL и BSD, а также разных версий GPL («GPLv2 or any higher»).

1.3 Почему свободности недостаточно

Как ни странно, но кроме религиозных, экономических и философских соображений на тему «почему всё должно быть свободным», существуют и вполне конструктивные измышления на эту тему «почему даже всё свободное не решает всех проблем».

Теорема 1.10. *Можно верить только тому программному обеспечению, которое написано лично вами.*

Thomphson compiler hack. Пусть у нас есть компилятор A и доступ к его исходным кодам (назовём их A'). Модифицируем A' в исходный код компилятора E (E') так, что E:

- распознаёт, когда ему на вход подаётся что-то похожее на A' и модифицирует необходимые части дерева разбора A', так, чтобы из него получалось дерево разбора E';
- распознаёт, когда ему на вход подаётся исходный код утилиты login (sshd, telnet, всё что угодно) и добавляет в неё, например, возможность логина в любого пользователя, если в качестве пароля написать «itisagooddaytodie».

После чего

- компилируем E' при помощи A, получаем исполняемый E;
- компилируем A' при помощи E, но получаем E;
- кладём в дистрибутив бинарный компилятор E и исходники A'.

Теперь, если кто-то вздумает скомпилировать утилиту login (или что мы там хотим испортить), то в результате система будет иметь backdoor, который не будет отображён ни в одних исходных кодах. Более того, даже при компиляции новой версии исходных кодов A', без какого-либо вмешательства со стороны автора E', с большой вероятностью получится новая версия компилятора E, со всеми новыми возможностями A, но с «интересными особенностями» при компиляции утилиты login. □

Заметим, что, в особо извращённом случае, даже дизассемблирование компилятора E может не спасти, потому как E может добавлять «цензуру» в исходники ядра, отвечающие за чтение бинарника E, или в работу с файловыми дескрипторами, через которые будет проходить дизассемблированный код E. Подобные же ухищрения, реализованные в аппаратуре, на практике вообще не обнаружить.

Глава 2

Приложения и система: низкий уровень

О низком уровне абстракции незримом, но важном.

2.1 ABI и системные вызовы

2.1.1 ABI

Определение 2.1. *API — Application Programming Interface. Не вдаваясь в подробности, можно считать, что это тот интерфейс, который видит разработчик.*

С API программисты встречаются ежедневно, в этом вопросе есть какие-то особенности, связанные с конкретными языками программирования (полиморфизм, инстанциация шаблонов, ...), но в курсе «операционных систем» нас это не очень интересует.

Определение 2.2. *ABI — Application Binary Interface. В отличии от API, ABI — это интерфейс, который интересует разработчиков компиляторов, ядер операционных систем и низкоуровневых библиотек.*

В ABI входят:

- *конвенции вызовов (C, Pascal, stdcall, fastcall, ...);*
- *соглашения о размещении данных в регистрах;*
- *соглашения о выравниваниях;*
- *организация переключений уровней привилегий, методы организации системных вызовов;*
- *соглашения о линковке;*
- *другие специфичные для текущей архитектуры вещи.*

Пройдёмся по перечисленному списку.

Конвенции вызовов — это правила, описывающие в каком порядке следует складывать аргументы вызываемой процедуры на стек и/или в регистры процессора, например, в стандартной конвенции языка C аргументы складываются на стек в обратном порядке, а семейство конвенций fastcall старается положить как можно больше аргументов в регистры процессора, а те, что не поместились — на стек.

Соглашения о размещении данных в регистрах обычно диктуются архитектурой процессора (например, на $x86_{32}$ многие арифметические операции размещают результаты вычислений в строго определённых регистрах) и необходимостью обратной совместимости с неподдерживаемым (legacy) кодом.

Где-то между конвенциями вызовов и соглашениями о размещении данных в регистрах находятся правила, регламентирующие какие регистры при выполнении вызова процедуры должна сохранять вызывающая, а какие — вызываемая сторона.

Следующим пунктом идут соглашения о выравниваниях, обычно диктуемые:

- архитектурой железа (например, процессоры ARM вообще не дают возможности обращаться к данным, не выравненным по размеру слова, а при хардварной виртуализации хозяйственные структуры данных обычно требуется размещать выравненными по размеру страницы памяти);
- используемыми протоколами;
- необходимостью совместимости структур данных с несколькими архитектурами одновременно (например, в системах, где одновременно используются разнородные вычислительные узлы);
- ...

Все озвученные выше вопросы нас не очень интересуют, поскольку они должны были быть подробно освещены в курсе по ассемблеру. Зато оставшиеся пункты говорят о том, что в операционных системах с разделением привилегий требуется реализация механизмов системных вызовов, а при наличии разделяемых библиотек следует договориться о том, как их создавать, подключать и использовать. Эти два вопроса нас так сильно интересуют, что мы рассмотрим их прямо сейчас.

2.1.2 Системные вызовы

Инструменты, предоставляемые для изоляции процессов друг от друга, могут сильно различаться между архитектурами. Например, глубокие знания по ассемблеру говорят нам, что (без учёта виртуализации) на процессорах семейства x86 для контекстов исполнения существует четыре уровня привилегий (два бита в селекторе сегмента кода), а на PowerPC — всего два.

У разработчиков операционных систем уровни привилегий почему-то принято называть «кольцами» (rings) и изображать их как показано на рис. 2.1.

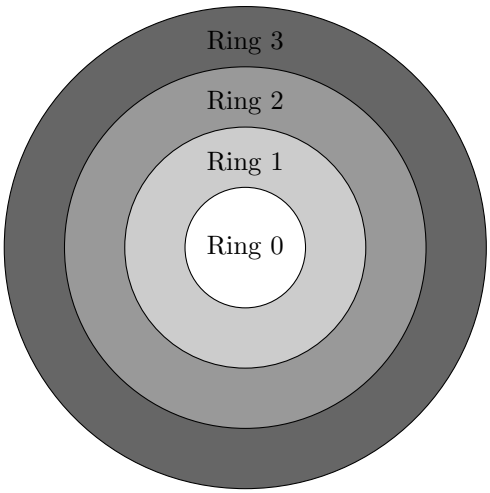

Рис. 2.1: Уровни привилегий — кольца

При этом считают, что в нулевом кольце («Ring 0») расположено ядро операционной системы, которому можно всё (читать и писать в любую область памяти, напрямую общаться со всем доступным железом), а в третьем кольце («Ring 3») — приложения пространства пользователя (aka «userspace», «юзерспейс»), у каждого из которых нет доступа ни к чему, кроме своей собственной области памяти.

Первое и второе кольцо (как нумеруют промежуточные уровни, если их больше двух — я не в курсе) представляют собой какие-то компромиссы между этими двумя крайностями.

В большинстве операционных систем (все UNIX-like, Windows, Haiku/BeOS, ...) используется только два уровня привилегий — ядро vs. юзерспейс, однако существовали и существуют системы, использующие больше (например, OS/2 использует три кольца, а Multics использовала восемь), или вообще использующие только одно кольцо (например, Inferno, Bluebottle). Популярность двухуровневой модели обусловлена простотой реализации и обширной поддержкой со стороны железа.

Остановимся на этом самом распространённом варианте и взглянем на мир с точки зрения программы, когда её код исполняется процессором в третьем кольце. Допустим, что этому коду понадобилось прочитать файл с жесткого диска. Процедуры, связанные с выполнением этой операции, требуют привилегий нулевого кольца для своей работы и находятся где-то в ядре, память которого также защищена от доступа из третьего кольца. Однако если рассматривать этот факт с точки зрения исполняемого кода, то ему нужно всего лишь на время поднять уровень своих привилегий. Очевидно, что если разрешить коду самовольно его менять и делать в нулевом кольце всё, что вздумается, то ни о какой «защищённости» речи быть не может. Но если вместе с изменением уровня привилегий исполняемая программа будет передавать управление коду, находящемуся в нулевом кольце, то этот код может проверить, а стоит ли делать то, что его просят, или это может плохо закончиться.

Определение 2.3. *Системный вызов (aka «syscall») — механизм передачи управления коду с повышенным уровнем привилегий, где вызываемая сторона, если требуется, проверяет легитимность запрашиваемых вызывающей стороной операций.*

Если проверка легитимности требуется, но не производится, или производится не достаточно тщательно, то такой код потенциально подвержен ошибкам нарушения прав доступа (access violation bugs), которые являются, пожалуй, самыми страшными угрозами безопасности системы (особенно для серверов с сервисами, предоставляющими shell-доступ (aka «удалённый терминал») или что-то в этом роде).

Но вернёмся к низкоуровневым механизмам. На столь любимом x86 существуют аж три метода передачи управления от кода с более низким уровнем привилегий коду в нулевом кольце (для тех, кто не помнит, что всё это значит, их краткие описания приводятся ниже):

- Interrupt Descriptor Table (IDT): Trap, Interrupt, Task (TSS);
- Global/Local Descriptor Table (GDT/LDT): CallGate;
- SYSCALL/SYSENTER.

И, как ни странно, все эти механизмы кем-нибудь да используются. Например, Linux для организации системных вызовов использует прерывания (однако на Pentium IV прерывания работают медленно, и вместо них используется SYSENTER), а FreeBSD использует CallGate. Не-x86 архитектуры обычно обходятся одними только прерываниями.

Теперь чуть подробнее об использовании упомянутых механизмов.

При использовании прерываний ядро при своей загрузке помещает в IDT дескриптор, обработчик которого представляет собой диспетчер системных вызовов. Аргументы системного вызова раскладываются вызывающим кодом в порядке, определяемом в соответствии с соглашениями ABI (на $x86_{32}$ это: eax, ebx, ecx, edx, esi, edi). В первый регистр обычно помещается номер системного вызова, во второй регистр — первый аргумент системного вызова, третий регистр — второй аргумент, и так далее, однако бывают системные вызовы, которые используют сразу два регистра под номер, а аргументы начинаются с третьего регистра. Если количество аргументов системного вызова больше, чем доступное количество регистров, то оставшиеся аргументы складываются на стек.

CallGate позволяет создать в адресном пространстве процесса специальную область, при far call на любую ячейку которой будет производится переключение из третьего в нулевое кольцо. Для приложения весь этот процесс выглядит как обычный вызов процедуры, а для ядра — как прерывание (ну почти).

Каждый вызов прерывания требует обращения к памяти, поскольку нужно найти дескриптор в IDT, чтобы достать оттуда сегмент с кодом обработчика. Инструкции SYSCALL и SYSENTER призваны сэкономить это обращение к памяти, благодаря использованию специальных регистров процессора (aka «model-specific registers», «MSR»), в которые ядро при своей загрузке помещает необходимую хозяйственную информацию (селекторы сегментов памяти обработчика системных вызовов). При этом сами инструкции SYSCALL/SYSENTER просто копируют данные из этих MSR в рабочие регистры. После этого разница между обработкой прерываний и SYSCALL/SYSENTER исчезает.

Теперь о том, что означает «обработчик представляет собой диспетчер системных вызовов». В самом начале исполнения обработчика прерывания, вызванного в целях совершения системного вызова, в регистрах процессора записан номер (или номера), идентифицирующий системный вызов, и (возможно не все) его аргументы. Чтобы суметь разобраться в том, что надо делать, ядро держит в своей памяти таблицу, в которой в n-ной ячейке находится адрес обработчика n-ого системного вызова (кстати, да, системные вызовы нумеруются с единицы). Обработчик прерывания просто передаёт управление по нужному адресу.

Как упоминалось ранее, может быть так, что первый регистр только указывает семейство системных вызовов, тогда по указанному адресу будет находиться новый диспетчер, руководствующийся в поиске обработчика числом из второго регистра. Если же был достигнут настоящий обработчик системного вызова, то он смотрит на нужные аргументы в регистрах (и заглядывает на стек, если необходимо), проверяет их корректность, а потом выполняет то, что просят. В семействе UNIX-like систем принято, что в случае успешного завершения системного вызова в первый регистр (на $x86_{32}$ — eax) будет помещено какое-то значение ≥ 0, а если всё плохо, то в этом регистре окажется значение $-$errno, где errno — номер ошибки. Как и с обычными процедурами, после завершения выполнения системного вызова управление будет возвращено на следующую за ним инструкцию.

Какие регистры будут сохранены ядром, а какие должно самостоятельно сохранить вызывающее приложение, опять же, диктуется ABI текущей архитектуры. На архитектурах, где доступно (относительно) большое количество регистров процессора, иногда просто выделяют «отрезок» регистров, зарезервированных ядром.

Поскольку прерывания и другие описанные механизмы требуют использования несколько отличного от работы с обычными процедурами языка C, то для всех (ну почти) системных вызовов стандартная библиотека (libc) предоставляет «обёртки». Такая обёртка над системным вызовом обычно делает следующее:

- сохраняет необходимые регистры процессора, которые могут быть изменены ядром;
- раскладывает идентификатор(ы) системного вызова и его аргументы по регистрам и на стек;
- вызывает прерывание, делает far call на CallGate или использует инструкции SYSENTER/SYSCALL;
- после того как управление будет возвращено, проверяет значение, записанное в первом регистре;
- если оно ≥ 0, то обёртка сама возвращает это значение;
- иначе — абсолютное значение заталкивается в переменную errno, а результатом вызова процедуры является -1.

Если системный вызов должен вернуть наружу более одного значения, то, как и с процедурами в языке C, оставшиеся значения будут записаны по соответствующим адресам, переданным в качестве его параметров (aka «передача по адресу»).

Рассмотрим один и тот же пример, реализованный на C и на ассемблере. Реализуем прекрасный «Hello, world!» с использованием двух системных вызовов: write и exit.

Системный вызов write записывает данные из буфера в файловый дескриптор, принимая три аргумента:

- файловый дескриптор, в который будет производиться запись;

- буфер, из которого будет производится запись;
- количество байт, которое нужно записать из буфера в файловый дескриптор.

Системный вызов exit завершает выполнение программы с кодом (aka «return value», «$?» в bash), указанным в качестве первого и единственного его аргумента.

Листинг 2.1: syscall.c

```
void main()
{
    write(1, "Hello, world!\n", 14);
    exit(0);
}

// gcc syscall.c
```

Вообще говоря, можно было бы сделать int типом возвращаемого значения main, и вместо вызова exit использовать обычный return с нужным кодом возврата (а в данном примере даже не меняя типа main заменить exit на return). Однако эти варианты эквивалентны, поскольку компилятор оборачивает вызов процедуры main служебной процедурой, которая и вызывает exit с результатом процедуры main в качестве аргумента (или с нулём, если main имеет тип void).

Следующее замечание заключается в том, что никакого системного вызова exit не существует, и процедура с таким именем есть только в стандартной библиотеке C. Системный вызов же носит имя _exit и его поведение немного отличается от того, что делает процедура exit, но об этих тонкостях речь пойдёт позже (см. разд. **??**).

Листинг 2.2 демонстрирует программу, почти эквивалентную программе 2.1, реализованную на ассемблере (в синтаксисе AT&T).

Первые две строки объявляют compile-time константы, обозначающие два рассмотренных ранее системных вызова. После чего начинается секция данных с двумя переменными: буфером со строкой приветствия миру и errno. Следующая инструкция говорит компилятору, что _start должна быть меткой, видимой из других единиц трансляции (в терминах C — не static процедура). После этого приведена реализация эквивалента процедура main из предыдущего листинга, с той разницей, что в errno попадает не абсолютное значение ошибки, а отрицательное (из соображений экономии места в листинге, чтобы он мог поместиться на один слайд презентации).

И последнее замечание: использовать системный вызов write так, как это делается в этих листингах, вообще говоря, нельзя, но об этих тонкостях тоже чуть позже (см. разд. **??**).

2.2 Компиляция, линковка, динамическая загрузка и связанные с этим вещи

2.2.1 Объектные файлы

В современном мире процесс превращения исходных кодов программы в её бинарное исполняемое представление (aka «компиляция»), в действительности состоит из двух этапов: компиляции и линковки. Немного порассуждаем о первом процессе, дабы понять, зачем нужен второй.

Во время компиляции (в её «классическом» варианте) компилятор конкретного языка программирования преобразует исходные коды программы в опкоды (op-codes) конкретного процессора (aka «бинарный код»). Если речь идёт о языках типа ассемблера или C, то полученный бинарный код для удобства дальнейшей с ним работы (как минимум в режиме отладки) должен быть очень похож на первоначальный исходный код, из которого его компилируют, в том смысле, что он точно также разбит на процедуры с такими же именами, что и в исходном варианте. Поскольку процессор в своей работе

Листинг 2.2: syscall.S

```
SYS_exit = 1
SYS_write = 4

.data
helloworld: .string "Hello, world!\n"
errno: .4byte

.globl _start
_start:
    movl    $SYS_write,%eax
    movl    $1,%ebx
    movl    $helloworld,%ecx
    movl    $14,%edx
    int     $0x80
    cmp     $0,%eax
    jl      oops
    jmp     ok

    oops:
    mov %eax, errno
    mov $-1, %eax

    ok: # %eax == result, $errno == -errno

    movl    $SYS_exit,%eax
    movl    $0,%ebx
    int     $0x80

# as -o syscall.o syscall.S
# ld syscall.o
```

использует исключительно адреса в памяти, а, например, отладчику нужны имена процедур (то есть строки), адреса которых лежат на стеке во время исполнения программы, то достаточно приложить к бинарному коду таблицу, сопоставляющую имена процедур их адресам, и все потребности обеих сторон будут удовлетворены.

Теперь, если вспомнить, что бывают не только сегменты кода с процедурами внутри, но и, например, сегменты данных, с которыми ассоциированы имена каких-то переменных (в программе на ассемблере этот факт заметнее, чем в программе на C), то было бы хорошо, чтобы все эти типы сегментов со всеми именами объектов в них можно было хранить вместе, но как-то различать.

Определение 2.4. *Объектные файлы — форматы файлов, позволяющие хранить вместе различные сегменты, представляющие собой результаты компиляции одного элемента трансляции.*

Таким образом, в объектных файлах располагаются все сегменты программы и соответствующие им таблицы с именами.

До тех пор, пока содержимое объектного файла всегда располагается в памяти по адресу, известному на этапе компиляции, ничего лишнего для запуска этого кода делать не надо, поскольку все адреса в нём указывают куда следует. Как только появляется желание размещать бинарный код программы «где-то» в памяти — сразу нужно что-то придумывать.

Самое наивное решение заключается в том, чтобы завести ещё одну табличку, которая сопоставляет в объектном файле всем инструкциям JMP, CALL, MOV и тому подобным имена процедур и переменных, на которые они ссылаются.

Не трудно предложить простой способ преобразования из объектных файлов в программы, которые можно запустить.

Алгоритм 2.5. *Алгоритм преобразования простых объектных файлов в код, который может исполнять процессор:*

- *договоримся о том, начиная с какого адреса в памяти будет располагаться программа;*
- *разместим значимое содержимое (не таблички с хозяйственной информацией, а только содержимое сегментов) объектного файла в памяти, начиная с этого адреса;*
- *пройдём по всем опкодам, делающими CALL, JMP или что-то подобное, и подставим в них настоящие адреса процедур, которые они вызывают (используя таблички имён процедур от сегментов кода и адрес начала программы);*
- *пройдём по всем опкодам, делающими MOV или как-то иначе работающими с ячейками памяти, и подставим в них настоящие адреса переменных, к которым они обращаются (снова используя соответствующие таблицы из объектного файла и адрес начала программы).*

После этого преобразования можно передать управление на адрес начала программы и она должна начать корректно работать.

Внутри объектных файлов иногда бывает удобно разделить один сегмент на несколько секций, например, для того, чтобы один сегмент кода мог быть составлен из частей, имеющих осмысленные имена. Например, в сегменте данных программы обычно присутствуют глобальные переменные, статические переменные, статические поля классов и другие тому подобные штуки.

2.2.2 Линковка и динамическая загрузка

Статическая линковка

Запасшись этими знаниями, рассмотрим два примера на языке C (листинги 2.3 и 2.4).

Договоримся, что каждый из этих двух файлов является отдельной единицей трансляции и они оба будут преобразованы в два разных объектных файла. Заметим, что первая единица трансляции определяет две процедуры: foo и main, а вторая — одну процедуру bar. При этом процедура main

Листинг 2.3: hipoteticTuA.c

```
int foo()
{
    return 1;
}

int main()
{
    return foo();
}
```

Листинг 2.4: hipoteticTuB.c

```
extern int foo();

int bar()
{
    return foo();
}
```

использует внутри себя foo, находящуюся в той же единице трансляции, а процедура bar использует foo, находящуюся в другой единице трансляции.

Преобразовать объектный файл, полученный из листинга 2.4, в бинарный код по предложенному ранее алгоритму 2.5 уже не удастся, поскольку процедура foo находится вне этого объектного файла. Зато не трудно реализовать алгоритм, который будет брать несколько объектных файлов и склеивать их в один большой (ресолвя перекрёстные ссылки на процедуры), который после этого можно преобразовывать в исполняемый код по алгоритму 2.5.

В самом простом случае линковщик занимается именно этим: он берёт несколько объектных файлов и склеивает их одноимённые секции, проверяя, чтобы не возникало конфликтов (например, одноимённых процедур в разных объектных файлах), и чтобы у всех используемых процедур была реализация хотя бы в одном объектном файле.

Заметим, что теперь может оказаться полезной возможность указывать в таблице с процедурами внутри объектного файла является ли каждая конкретная процедура видимой снаружи данного объектного файла или нет (кстати, в коде на C требуемый эффект можно получить при помощи ключевого слова «static»).

Динамическая линковка

Теперь представьте, что процедура foo из листинга 2.3 настолько часто используется в разных программах, что разработчики решили вынести её в отдельную единицу трансляции, которую назвали «программной библиотекой». Со временем в эту программную библиотеку стали добавлять и другие полезные процедуры. В результате, размер объектного файла у этой библиотеки стал больше суммы размеров всех объектных файлов среднестатистической программы. Всё бы ничего, но при каждой линковке любой среднестатистической программы с этой прекрасной библиотекой, размер результирующего бинарника будет становится в два раза больше. С учётом того, что обычно программа не использует все процедуры из библиотеки, то и вся библиотека обычно не нужна, но прилинковывать её придётся всё-таки всю.

Можно попытаться разделить большую библиотеку на несколько независимых поменьше, но, во-первых, это не всегда возможно, а, во-вторых, есть и другая проблема с предложенным типом линковки: если вдруг в библиотеке будет обнаружен баг, то, после его исправления, все программы, её использующие, придётся с ней как минимум перелинковывать, а то и целиком всё перекомпилировать.

Значит нужен какой-то более продвинутый метод линковки. Заметим, что в этом продвинутом методе линковки просто склеивать все используемые объектные файлы нельзя, а нужно уметь линковаться с внешними объектными файлами, то есть объектные файлы должны уметь ссылаться на процедуры в других объектных файлах. С другой стороны, поскольку теперь, при запуске программы, в память нужно загрузить несколько объектных файлов, а загружаемые объектные файлы, в свою очередь, тоже могут ссылаться на другие объектные файлы, то и метод загрузки должен стать несколько сложнее.

Таким образом, продвинутый метод линковки разваливается на два отдельных процесса:

- саму линковку — склеиваются объектные файлы, которые можно склеить, и производятся проверки валидности вызовов между объектными файлами, которые не следует склеивать;
- и динамическую загрузку — связанные объектные файлы загружаются в память и между ними восстанавливаются нужные ссылки.

Ещё раз осмыслите всё то, что написано выше, и поедем дальше.

Представим себе, что всё содержимое всех объектных файлов, необходимых для работы программы, уже как-то разложено в памяти. Для того, чтобы процедура, находящаяся в одном объектном файле, вызвала процедуру из другого объектного файла, первой из них нужен адрес второй. Если разложение объектных файлов в памяти от одного запуска программы к другому может меняться (а обычно это именно так), то вычислить этот адрес заранее (на этапе линковки) невозможно.

Методы, доступные динамическому загрузчику для обхода этой неприятности, отличаются на различных архитектурах. Например, на x86_{32} отсутствует возможность длинной относительной адресации, и все адреса в инструкциях CALL должны быть абсолютными, поэтому после загрузки программы в память приходится пройтись по всем этим инструкциям и записать в них абсолютные адреса вызываемых процедур. Точно также во все инструкции обращения к переменным следует записать их абсолютные адреса.

Если реализовывать это поведение «в лоб», то при каждом запуске программы нужно выполнить уйму работы: для каждого вызова процедуры в каждом объектном файле нужно найти в остальных объектных файлах процедуру, на которую он ссылается, а потом ещё раз примерно тоже самое для общих глобальных переменных. Попробуем что-то сэкономить, заметив, что:

- число вызовов процедур всегда больше, чем число процедур, которые вызывают;
- аналогичное утверждение верно и для переменных;
- для ресолвинга вызовов и обращения к переменным, расположенным внутри текущего объектного файла, не нужно много работы.

Последнее утверждение может не казаться столь очевидным, как первые два.

На архитектурах с доступной длинной относительной адресацией (например, x86_{64}) для обращения к ресурсам текущего объектного файла вообще никакого ресолвинга не нужно, поскольку все относительные адреса можно вычислить на этапе компиляции. На архитектурах без таких возможностей используется следующий трюк.

Алгоритм 2.6. *Скомпилируем код так, как будто в памяти он будет расположен начиная с нулевого адреса. Тогда для того, чтобы при расположении кода в памяти, начиная с адреса N, сделать все адреса внутри инструкций в коде валидными, нужно просто к каждому адресу в каждой инструкции прибавить N.*

Таким образом, зная адрес начала объектного файла в памяти и список смещений (относительно начала этого объектного файла) всех интересующих нас адресов внутри инструкций JMP, CALL, MOV и тому подобных, не трудно (относительно) быстро сделать все локальные адреса валидными.

Код, скомпилированный таким образом, принято называть «релокабельным» (relocable), а процесс, описанный в алгоритме **??** — «релокацией».

Но что делать с внешними вызовами? Вспомнив предыдущие три пункта со свойствами обращений к памяти, нетрудно догадаться, что следует завести «косвенную» таблицу.

Для вызовов процедур в ней можно расположить инструкции JMP с абсолютными адресами внешних процедур, тогда вызов внешней процедуры внутри объектного файла будет выглядеть как вызов локальной процедуры по адресу ячейки с соответствующим JMP'ом в таблице, а адреса в самой таблице могут быть заполнены на этапе динамической загрузки.

Для обращений к общим глобальным переменным можно поступить аналогичным образом, с той разницей, что в таблице придётся хранить адреса переменных, а не инструкции для доступа к ним, а всю адресацию к этим переменным внутри объектного файла делать косвенной.

Пример возможного расположения в памяти трёх объектных файлов приведён на рис. 2.2. У первого и третьего объектного файлов изображены сегменты:

- таблица внешних вызовов, где каждый «JMP» — ассемблерная инструкция JMP (с адресом, записанным туда динамическим загрузчиком);
- таблица внешних данных, где каждый «ADDR» — адрес внешних данных, записываемый в эту ячейку динамическим загрузчиком;
- сегмент данных (с, например, глобальными переменными);
- сегмент кода, где «. . . » — это какие-то ассемблерные инструкции, а «CALL» — ассемблерная инструкция CALL (заметим, что на архитектуре с доступной относительной адресацией динамический загрузчик может и не менять адреса внутри этих инструкций).

Таким образом, каждая ячейка, помеченная «JMP» или «CALL», является одной ассемблерной инструкцией, а ячейки «. . . DATA . . . » и «. . . » могут содержать внутри целые промежутки с данными/исполняемым кодом.

Если множество объектных файлов с подключаемыми программными библиотеками в системе статично, и они все могут позволить себе при каждом запуске каждой программы располагаться по статическим адресам, то, вообще говоря, можно заранее вычислить адреса расположения в памяти для всех библиотек, применить к каждой из них алгоритм 2.6, и больше не делать этого при динамической загрузке. Однако такие строгие условия в современных системах практически никогда не выполняются, а при первом же изменении множества программных библиотек всю работу придётся делать заново (не говоря уже о том, что на 32-х битной архитектуре может просто не хватить адресного пространства для того, чтобы разместить в нём все доступные библиотеки).

Ещё раз осмыслите всё то, что написано выше, и поедем дальше.

Итого, типичный сценарий жизни обычной программы выглядит следующим образом:

- компилируем: берём исходные коды, получаем объектные файлы;
- линкуем: склеиваем некоторые объектные файлы в один, а некоторые помечаем для динамического склеивания при загрузке;
- в результате получаем один или несколько объектных файлов, которые можно запускать (aka «бинарники»);
- при запуске бинарника, динамический загрузчик раскладывает в памяти все объектные файлы, от которых он зависит, а также объектные файлы, от которых зависят нужные этому объектному файлу объектные файлы, а также объектные файлы, от которых зависят объектные файлы, от которых зависят объектные файлы, нужные . . . и т.д.;
- если требуется, то ко всем адресам внутри объектного файла добавляется адрес его начала в памяти;
- динамический загрузчик восстанавливает все связи между объектными файлами, заполняя соответствующие таблички;

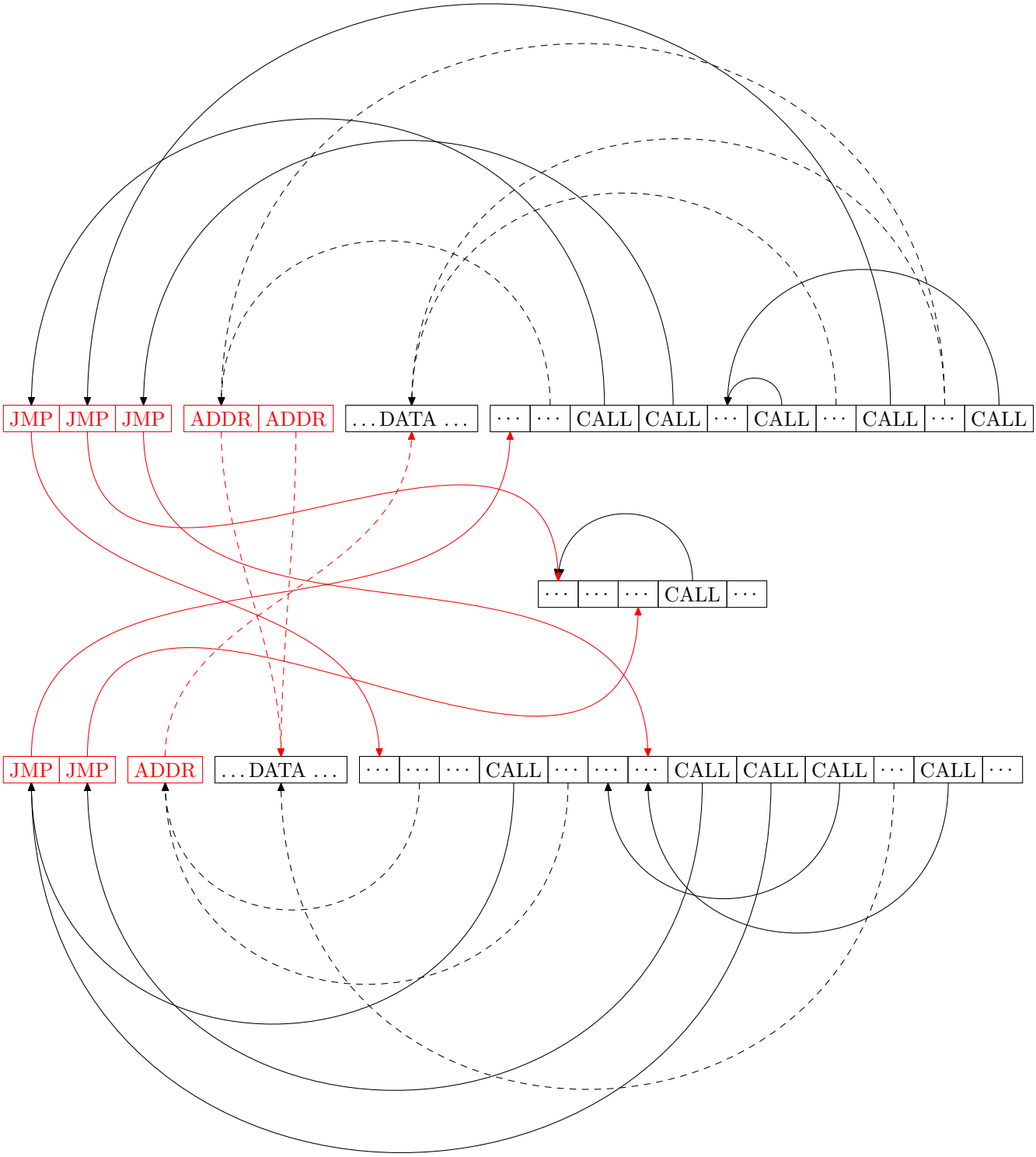

Рис. 2.2: Расположение трёх объектных файлов в памяти (с восстановленными при помощи динамического загрузчика связями). Пунктирными стрелками обозначены обращения к данным (глобальным переменным), сплошными — вызовы процедур. Красным выделено всё то, до чего не может не дотронуться динамический загрузчик.

- управление передаётся загруженной программе.

Конечно, в принципе, от таблицы внешних вызовов можно было бы отказаться, если адреса внешних процедур хранить в таблице внешних данных, однако тогда вызовы процедур внутри кода объектного файла перестанут быть похожими на вызовы процедур, и, кроме того, перестанут работать некоторые полезные трюки (смотри разд. 2.2.2). На практике же, в связи с тем, что таблицу внешних вызовов требуется размещать в исполняемом сегменте кода (а потому надо бы запретить её модифицировать), а вычисление абсолютных адресов внешних процедур может производиться не во время динамической загрузки, а откладываться (опять же смотри разд. 2.2.2 ниже), в таблицу внешних данных помещают всё-таки и адреса внешних процедур. После чего она приобретает название «Global Offset Table» (aka «GOT»). Однако таблицу внешних вызовов, в отличие от предложения выше, не выкидывают, а размещают в ней косвенные JMPы по адресам, записанным в соответствующих ячейках GOT. Сама такая таблица приобретает название «Procedure Linkage Table» (aka «PLT»).

Если теперь расстояние между PLT и GOT вычислить на этапе компиляции, то в ячейки PLT можно записывать адреса соответствующих ячеек GOT так, как это раньше делалось для релокабельного кода (представим, что объектный файл располагается в памяти с нулевого адреса . . .). Таким образом, PLT также становится релокабельной, а «умное» заполнение на этапе динамической загрузки требуется только для GOT.

PIC

В реальной жизни, в целях повышения безопасности, часто применяется механизм рандомизации памяти, когда динамический загрузчик каждый объектный файл, по возможности, располагает в памяти по случайному адресу. Очевидно, что при этом перестаёт работать кеширование заранее вычисленных адресов и другие подобные штуки.

Править только GOT (и PLT) при каждой загрузке объектника в память не есть очень затратно (поскольку обычно эти таблицы очень маленькие, и, кроме того, в системе с рандомизацией всё равно от них никуда не деться). С другой стороны, суммирование адреса начала объектника почти с каждой его инструкцией (в реальных программах инструкции по работе с памятью встречаются очень часто) — занятие достаточно трудоёмкое. Поэтому на время оставим таблички и посмотрим на код внутри каждого сегмента.

Определение 2.7. *Placement Independent Code (PIC) — код, корректная работа которого не зависит от его расположения в памяти.*

Архитектуры с длинной относительной адресацией снова оказываются чертовски удобны, поскольку бинарный код для них без труда приводится к требованиям PIC и не требует релокации при загрузке. С другой стороны, очевидно, что без какого-то аналога относительной адресации получить код, независимый от абсолютных адресов невозможно. Возникает вопрос, а можно ли как-нибудь сэмулировать относительную адресацию на архитектуре, где таких возможностей нет?

Оказывается это не так уж и трудно сделать. Рассмотрим следующий ассемблерный код, представленный на листинге 2.5.

Листинг 2.5: pic.S

```
call foo
foo:
pop %eax
```

Идея этого великого кусочка кода заключается в том, что, для того, чтобы такой код выполнить, на этапе динамической загрузки необходимо записать абсолютный адрес foo только в одну инструкцию CALL, а после исполнения этого кода в регистре eax оказывается абсолютный адрес текущей инструкции. В рантайме же, имея абсолютный адрес одной инструкции и смещения от неё до всех других

инструкций (вычисленные на этапе компиляции), можно вычислить абсолютный адрес любой другой инструкции.

Чтобы несколько упростить себе жизнь, связанную с этой арифметикой, как и ранее используется GOT, заполняемая на этапе динамической загрузки. Тогда после получения адреса текущей инструкции (как в листинге 2.5), вычисленный адрес GOT сохраняют в каком-то регистре (на $x86_{32}$ — в ebx), после чего обращения ко всем переменным и процедурам (PLT обычно располагается рядом с GOT, а потому её адрес нетрудно вычислить имея адрес GOT) производятся через эти таблицы. Заметим, что в отличие от не-PIC кода, в GOT теперь будут хранится не только внешние глобальные переменные и процедуры, как это было раньше, а вообще все переменные, не живущие на стеке или в регистрах, а также адреса всех процедур, в том числе и локальных.

Писать такой ассемблерный код руками весьма непросто, зато любому двухпроходному компилятору это прекрасно удаётся.

Весьма условная схема объектного файла, скомпилированного под PIC с поддержкой ленивой динамической линковки представлена на рис. 2.3.

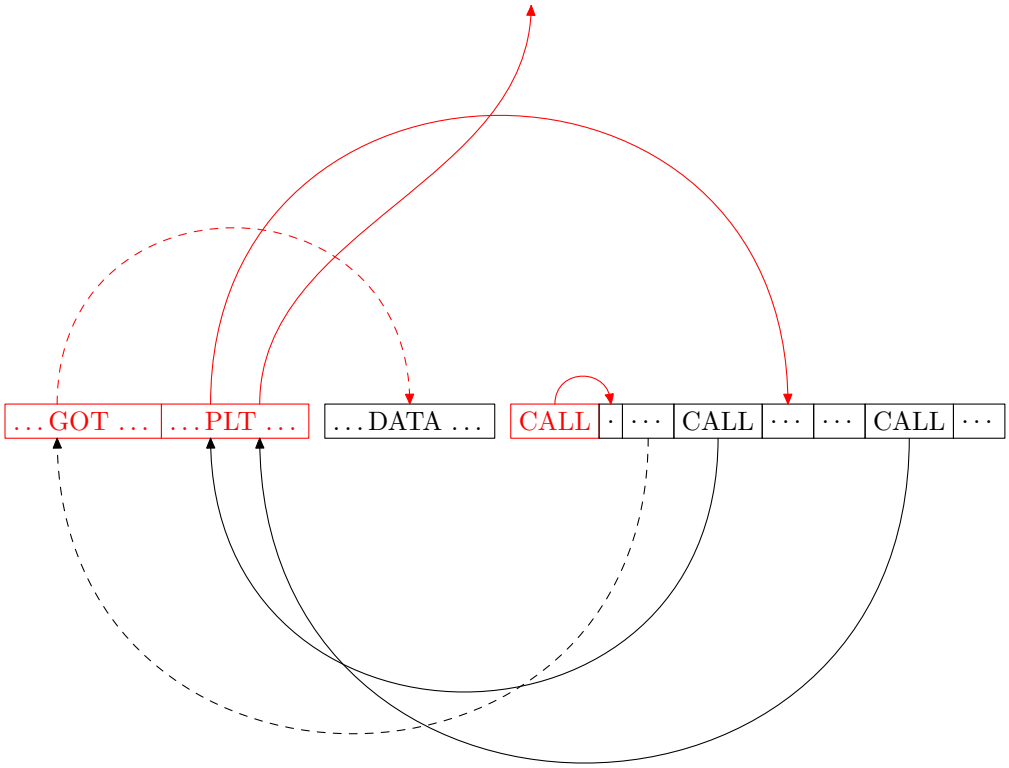

Рис. 2.3: PIC в действии. Стрелкой «в никуда» обозначен внешний вызов

В большинстве дистрибутивов GNU/Linux все shared-object'ы (программные библиотеки) компилируются в PIC. В некоторых особенно параноидальных дистрибутивах даже обычные программы компилируются в PIC, чтобы вообще всё располагать по случайным адресам в памяти.

Ленивость и другие забавные штуки

Предложенная модель с выделенной GOT позволяет на этапе динамической загрузки подменять адреса переменных и процедур, к которым будет обращаться загружаемый объектный файл, на адреса не тех переменных и процедур, которые он запрашивает (поскольку заполнение GOT всецело находится в руках динамического загрузчика). Сохранение выделенной PLT даже при условии того, что все адреса

хранятся в GOT, также предоставляет немало свободы действий, поскольку если вместо одного JMP на нужный адрес размещать для каждого внешнего вызова в PLT несколько инструкций, то можно перед запуском требуемой процедуры делать разного рода забавные вещи.

Очевидно, то оба варианта игр с GOT и PLT могут нарушить нормальный порядок работы программы, однако их аккуратное использование может находить весьма удачные применения. Например:

- при загрузке объектного файла, можно все вызовы обёрток системных вызовов из стандартной библиотеки, работающих с сетью, заменить на вызовы процедур, прозрачно направляющих все запросы на какой-нибудь прокси-сервер (таким образом, для проксирования сетевого трафика отдельных программ (или даже всей системы) в самих программах ничего менять не нужно, кроме того, можно бесплатно получить централизованное конфигурирование сетевых настроек);
- весьма часто можно встретить подмену вызовов malloc и free на аналоги, ведущие статистику (для проверки кода на утечки памяти);
- в особо извращённых случаях вызовы типа fsync заменяют на заглушки, которые ничего не делают (повышает скорость работы программы, но, очевидно, всякие гарантии на согласованность данных на диске пропадают).

При использовании нескольких инструкций на элемент в PLT, дополнительные инструкции можно использовать для отладки (вставлять перед вызовом требуемой процедуры CALLы, производящие отладочные операции) или для ленивой динамической линковки. На последнем остановимся подробнее.

Очень часто, при нормальном исполнении программы, некоторые внешние вызовы (например, процедуры вызываемые в обработчиках ошибок) вообще никогда не производятся, тем более не производится вызовов процедур, которые потенциально могут быть вызваны из процедур, которые сами не были вызваны, и так далее. Поэтому иногда можно существенно сэкономить, если реализовать возможность ресолвинга адресов внешних процедур не при динамической загрузке объектного файла, а во время исполнения программы при первом обращении к запрашиваемой внешней процедуре.

Таким образом, идея ленивой линковки заключается в следующем:

- при вызове записи из PLT проверяется, что адрес вызываемой внешней процедуры уже известен, после чего на него делается JMP;
- в случае, если адрес ещё не известен, то производится вызов процедуры динамического загрузчика (с идентификатором требуемого внешнего вызова в качестве параметра);
- динамический загрузчик находит абсолютный адрес требуемой процедуры, сохраняет его в месте, доступном проверке из PLT, после чего делает JMP на этот адрес.

В GNU/Linux для реализации динамической линковки, каждый элемент в PLT содержит три инструкции (листинг 2.6):

- JMP по адресу, записанному в соответствующей записи GOT (как отмечалось ранее, для каждой записи в PLT присутствует соответствующая запись с адресом процедуры в GOT);
- PUSH идентификатора требуемого вызова;
- JMP на процедуру динамического загрузчика.

Соответствующие элементы GOT изначально содержат адреса второй инструкции в соответствующей записи PLT (инструкция PUSH ...).

Таким образом, при первом вызове первая инструкция записи в PLT сделает JMP на вторую инструкцию в этой же записи PLT. Эта инструкция положит на стек адрес соответствующего поля в GOT, после чего следующая инструкция обратится к динамическому загрузчику. При втором и последующих вызовах этой записи в PLT сразу будет произведён косвенный JMP на адрес необходимой процедуры.

Ещё раз осмыслите всё то, что написано выше, и поедем дальше.

Листинг 2.6: link-lazy-plt.S

```
jmp *GOT+n
push #offset
jmp DINLN
```

Типы линковки

Классифицируя упомянутые выше методы можно выделить следующие типы линковки:

- статическая — когда несколько объектных файлов склеиваются в один;
- динамическая — когда собственно восстановление ссылок переносится на этап динамической загрузки, этот тип в свою очередь подразделяется на:
  - модификация всех адресов в инструкциях прямо в коде;
  - GOT, релоцируемая PLT, релоцируемый код;
  - GOT, релоцируемая PLT, PIC;
  - GOT, релоцируемая PLT, релоцируемый код, ленивая динамическая загрузка;
  - GOT, релоцируемая PLT, PIC, ленивая динамическая загрузка;
  - GOT, относительная PLT, бесплатный PIC (на архитектурах с длинной относительной адресацией);
  - GOT, относительная PLT, бесплатный PIC, ленивая динамическая загрузка (на архитектурах с длинной относительной адресацией);
- статическая линковка динамических объектов.

Последний тип линковки представляет собой вариант динамической линковки с кешированием, когда все объектные файлы всегда располагаются по статическим адресам в памяти. Однако, опять же, стоит только одному из этих файлов измениться, как все объектные файлы всей системы нужно будет перелинковать.

2.2.3 ELF

Исторически существовало (а где-то существует и до сих пор) множество различных форматов объектных и исполняемых файлов. На сегодняшний день самыми распространёнными являются: COFF, PE и ELF, однако тенденция к унификации медленно движет мир в сторону последнего, поскольку он поддерживается на почти всех распространённых архитектурах, но, в тоже время, является достаточно простым внутри. Приятным преимуществом ELF перед некоторыми другими форматами является то, что в него «умещаются»:

- объектные файлы, используемые во время компиляции и линковки;
- результирующие бинарные файлы;
- файлы программных библиотек;

в то время как у многих других форматов внутренняя структура этих типов объектных файлов сильно различается. Кроме того, с 1999 года ELF является стандартом объектных и исполняемых файлов UNIX-like систем для x86.

Внутренняя структура ELF (который, кстати, расшифровывается как «Executable and Linkable Format») файла представлена на рис. 2.4. После заголовка ELF со всякой хозяйственной информацией следует таблица сегментов (aka «segments table», «program header table»). Сразу за этой таблицей

начинается «тело» ELF, за которым может располагаться таблица секций. Последняя таблица *может располагаться* по той причине, что она, по большому счёту, нужна только линковщику, а потому (в целях экономии места) может быть почти безболезненно отрезана от ELF исполняемого бинарника.

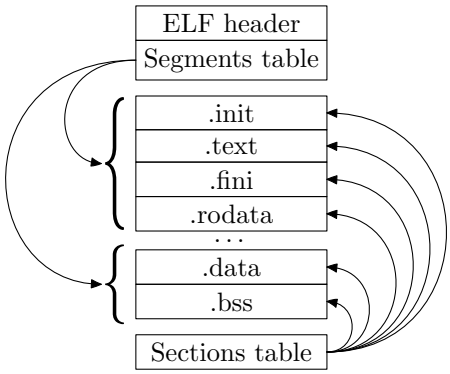

Рис. 2.4: Внутренняя структура ELF

Таблица сегментов состоит из строк, каждая из которых описывает один сегмент. Описание сегмента содержит (приводятся интересующие нас далее поля):

- тип сегмента;
- смещение относительно начала тела ELF;
- адрес в виртуальном адресном пространстве, по которому этот сегмент нужно будет разместить;
- адрес в физическом адресном пространстве, по которому этот сегмент нужно будет разместить;
- размер сегмента в ELF;
- размер сегмента в памяти;
- флаги.

Например, для /bin/bash эта таблица имеет вид, представленный в листинге 2.7.

Листинг 2.7: Сегменты ELF /bin/bash

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
    [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD           0x000000 0x08048000 0x08048000 0xbd674 0xbd674 R E 0x1000
LOAD           0x0bd674 0x08106674 0x08106674 0x04790 0x0959c RW  0x1000
DYNAMIC        0x0bd688 0x08106688 0x08106688 0x000e0 0x000e0 RW  0x4
NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
GNU_EH_FRAME   0x0bd5b8 0x081055b8 0x081055b8 0x0002c 0x0002c R   0x4
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

Таблица секций похожа на таблицу сегментов, но с несколько более сложной структурой, содержащей (снова приводятся только интересующие нас далее поля):

- имя секции;
- тип секции;

- адрес в виртуальном адресном пространстве;
- смещение относительно начала тела ELF;
- размер секции.

Для /bin/bash эта таблица имеет вид, представленный в листинге 2.8.

Листинг 2.8: Секции ELF /bin/bash (отрывок)

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	08048134	000134	000013	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	08048148	000148	000020	00	A	0	0	4
[3]	.note.gnu.build-i	NOTE	08048168	000168	000024	00	A	0	0	4
[4]	.**hash**	HASH	0804818c	00018c	0040c4	04	A	6	0	4
[5]	.gnu.**hash**	GNU_HASH	0804c250	004250	0035f0	04	A	6	0	4
[6]	.dynsym	DYNSYM	0804f840	007840	0082a0	10	A	7	1	4
[7]	.dynstr	STRTAB	08057ae0	00fae0	007e94	00	A	0	0	1
[8]	.gnu.version	VERSYM	0805f974	017974	001054	02	A	6	0	2
[9]	.gnu.version_r	VERNEED	080609c8	0189c8	0000a0	00	A	7	2	4
[10]	.rel.dyn	REL	08060a68	018a68	000040	08	A	6	0	4
[11]	.rel.plt	REL	08060aa8	018aa8	0005c8	08	A	6	13	4
[12]	.init	PROGBITS	08061070	019070	000030	00	AX	0	0	4
[13]	.plt	PROGBITS	080610a0	0190a0	000ba0	04	AX	0	0	4
[14]	.text	PROGBITS	08061c40	019c40	08a1ac	00	AX	0	0	16
[15]	.fini	PROGBITS	080ebdec	0a3dec	00001c	00	AX	0	0	4
[16]	.rodata	PROGBITS	080ebe20	0a3e20	019798	00	A	0	0	32

Таким образом, и сегменты, и секции указывают на отрезки внутри объектного файла в формате ELF. Обычно подразумевается, что каждый сегмент содержит в себе несколько секций целиком, но на практике одна секция может пересекать несколько сегментов сразу.

Именование файлов в ELF формате, в принципе, ничем не ограничено, но для промежуточных объектных файлов, генерируемых в процессе компиляции принято использовать расширение «.o», а программные библиотеки размещать в файлах с именами начинающимися на «lib» и с расширением «.so».

2.2.4 Секции и сегменты

Тут будет от интерпретации представленных выше структур.

2.2.5 Линковка

Тут будет о линковке ELFок. В данный момент очень вяло написано.

Всё это очень интересно, но что всё же собственно делает линковщик с объектными файлами в формате ELF? Оказывается, он просто склеивает одноимённые секции в объектных файлах, линкуемых статически и проверяет корректность внешних вызовов к объектным файлам, линкуемым динамически.

При этом порядок, в котором будут склеены объектные файлы, вообще говоря, имеет значение. Например, существует несколько стандартных объектных файлов, статически прилинковываемых (почти) к любой программе:

- crti.o — содержит определения процедур _init и _fini;
- crtbegin.o и crtend.o — (для C++) добавляют в тела процедур _init и _fini вызовы глобальных конструкторов и деструкторов соответственно;

- crtn.o — содержит инструкции RET для процедур _init и _fini.

Для каждого объектного файла динамический загрузчик, после размещения объектного файла в памяти, но до запуска процедуры _start, вызывает процедуру _init (если она определена). Линковка объектных файлов программы с этими объектными файлами требует определённого порядка: crti, crtbegin, объектные файлы программы, crtend, crtn.

2.2.6 Динамический загрузчик

Снова вспомним, что, не смотря на всю внешнюю простоту происходящего, в запуске объектного файла участвуют трое:

- пользователь, который его запускает, то есть, например, я (а вернее — некоторая программа от моего имени), набирая в консоли «/bin/bash»;
- ядро, которому приходится обрабатывать системный вызов exec с «/bin/bash» в качестве какого-то параметра;
- и, внимание, *динамический загрузчик*, который будет вызван ядром, которому очень хочется обработать exec с «/bin/bash» в качестве какого-то параметра, но не очень хочется разбираться во всех этих премудростях из сегментов, секций, программных библиотек, PIC и тому подобного.

Иначе говоря, после того, как ядро разберётся в том, что его просят запустить, и в том, можно ли это запустить, вместо того, чтобы самостоятельно разбираться во всех возможных сценариях запуска программы, оно запускает динамический загрузчик, каким-то образом указывая ему на нужный ELF (передаёт путь к файлу в качестве аргумента argv или передаёт открытый файловый дескриптор, связанный с ним). Поскольку сам динамический загрузчик является программой в ELF, то ядру всё-таки приходится что-то понимать во внутренностях этого формата, однако класс ELF'ок, которые способно загрузить ядро, очень ограничен.

Итого, в UNIX-like системах на самом деле обычно есть *два* динамических загрузчика: один в ядре, и ещё один в юзерспейсе. Причём сил первого достаточно на очень простые ELF'ки, а второй умеет очень много, хотя сам он скомпилирован в простую ELF'ку.

Не смотря на то, что в ядре тоже есть кусок кода, выполняющий роль динамического загрузчика, когда говорят «динамический загрузчик» обычно подразумевают именно второй.

В GNU/Linux динамический загрузчик располагается в файловой системе по адресу «/lib/ld-linux.so.$mversion», где «$mversion» — мажорный номер его версии. Этот файл, в свою очередь, может являться симлинком на что-то вроде «/lib/ld-$version.so», где «$version» — полный номер версии загрузчика.

При загрузке объектных файлов ld-linux ищет связанные объектные файлы (библиотеки) в директориях «/lib», «/usr/lib» и «/usr/local/lib» в соответствующем порядке. Если по каким-то причинам этот порядок нужно изменить или добавить туда ещё какую-то директорию, то в окружение следует добавить переменную «LD_LIBRARY_PATH» со значением, содержащим необходимые пути, перечисленные через запятую. Тогда динамический загрузчик заглядывает в пути, указанные в этой переменной, а уже потом идёт по стандартным.

2.3 Практика

Тут будет о том как делать .so, вызывать динамический линковщик в райнтайме, делать программы с плугинами и другие интересные вещи.

2.4 Комментарии

Из текста не совсем понятно соотношение между «объектными файлам», «бинарниками» и «программными библиотеками». Автору не совсем понятно как это понятно объяснить, ибо границы размыты.

2.5 Краткое содержание

Тут будет краткое содержание написанного выше.