Визитка

ДЕНИС СИЛАКОВ, к. ф.-м. н., член рабочей группы LSBстарший архитектор ЗАО «РОСА», занимается автоматизацией разработки ОС «РОСА»

Стандартизация в мире Linux
Зачем и как?

Сегодня существует не одна сотня дистрибутивов Linux — каждый со своей спецификой. Как им удается избежать несовместимости?

Разработка ПО в мире Linux идет очень быстро, ведь она в существенной степени ведется силами энтузиастов, работающих ради интереса и привносящих в продукты действительно новые идеи. При этом многие проекты придерживаются принципа частых релизов (Release early, release often), описанных в ставшей классикой в мире свободного ПО книге «Собор и базар» Эрика Реймонда, когда новые версии программ выпускаются даже после небольших изменений в коде.

Политика частых релизов удобна в плане привлечения пользователей к тестированию – чем больше людей попробуют новую версию, тем больше вероятность выявить содержащиеся в ней ошибки. Однако у этого подхода есть и серьезные недостатки.

На всех проблемах мы останавливаться не будем, а затронем наиболее важную для разработчиков, опирающихся на сторонние программы, а именно проблему совместимости различных версий одного и того же программного продукта. Ведь побочным эффектом политики частых релизов является появление множества версий одной и той же программы или библиотеки, не всегда совместимых друг с другом. Помноженная на многообразие дистрибутивов Linux такая проблема становится серьезным препятствием для тех, кто хочет создавать приложения, не привязанные к конкретным вариациям этой ОС.

Одним из способов упорядочить такое многообразие является стандартизация – выработка некоторых правил, общих для всех дистрибутивов (или по крайней мере для большинства из них).

В современном мире Linux процессы стандартизации затрагивают две основные области:

внешние интерфейсы, предоставляемые компонентами дистрибутивов сторонним программам, – в первую очередь библиотечные функции, а также другие способы взаимодействия приложений с системой (опции утилит командной строки, средства межпроцессного взаимодействия и другие);

способы интеграции приложения в ОС – если приложения нет в репозиториях дистрибутива, то его надо каким-то образом установить в систему (разместив файлы программы в файловой системе согласно правилам дистрибутива, добавив приложение в меню, проассоциировав с ним нужные файлы и т.д.), а также гарантировать, что приложение не нарушает в процессе своей работы никаких правил, принятых в системе.

Начнем по порядку – со стандартов на внешние интерфейсы. Для начала давайте познакомимся с фундаментальной концепцией, лежащей в основе таких стандартов, – обратной совместимостью.

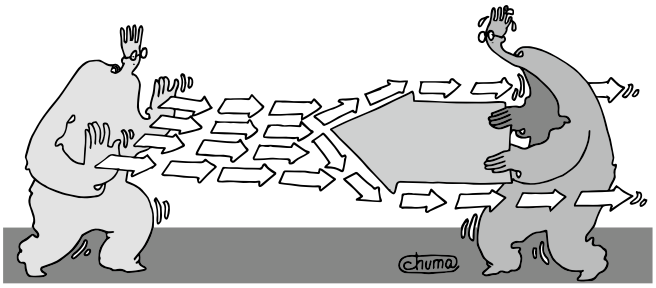
Обратная совместимость

В общей формулировке термин «обратная совместимость новой версии программного компонента со старой» означает, что все внешние интерфейсы, присутствующие в старой версии, присутствуют и в новой. Как следствие, если какое-то приложение работало со старой версией этого компонента, то оно будет работать и с новой версией. Возможно, такое определение звучит несколько абстрактно; давайте посмотрим, что оно означает в реальной жизни, и почему обратная совместимость для Linux является очень актуальным вопросом.

Предположим, что вы используете в своей программе на языке C некоторую стороннюю библиотеку libfoo и вызываете ее функцию, которой передаете аргумент – целое число – func(int x). А теперь допустим, что разработчик библиотеки в новой версии решил, что одного аргумента функции недостаточно и надо передавать два целых числа – func(int x, int y). Казалось бы, ничего страшного – небольшое изменение в программе, и она отлично работает с новой версией библиотеки.

Вспомним, однако, тот факт, что в мире нет некоторого «единого» или «главного» Linux. Вместо этого есть несколько сотен независимых (в той или иной степени) дистрибутивов, развивающихся по своим правилам и имеющих собственные жизненные циклы.

Если вы хотите, чтобы ваша программа стала популярной среди пользователей Linux, вам необходимо обеспечить ее работу в как можно большем числе вариаций этой систе-



Разработчики в мире Linux — энтузиасты, работающие ради интереса и привносящие в продукты новые идеи

мы. Конечно, можно надеяться на мэйнтейнеров дистрибутивов, но сначала вы должны их заинтересовать своим детищем; к тому же мэйнтейнеры оценят, если сборка вашего приложения не займет у них много времени.

Допустим, что библиотека libfoo достаточно распространена и есть во всех дистрибутивах. Но какая версия этой библиотеки присутствует в конкретной системе – старая или новая? Вполне возможно, что обе версии успели разойтись достаточно далеко, и теперь вам необходимо либо распространять две версии своей программы (а пользователям – определять, какая именно версия им подходит), либо применять ряд трюков в коде программы и уже в процессе работы определять, какие аргументы необходимо передать функции func() в используемой версии библиотеки.

Конечно, можно поставлять фиксированную версию libfoo как часть приложения или статически слинковать программу с этой библиотекой. Но (помимо увеличения размера приложения), у такого способа есть существенный недостаток – вам придется отслеживать и исправлять ошибки не только в своем коде, но и в коде libfoo. В частности, если в libfoo обнаружится уязвимость, то ваше приложение будет представлять угрозу для пользователя, которую надо будет оперативно устранять.

Поэтому даже небольшое изменение одной библиотечной функции сулит немало неприятностей разработчикам. Что уж говорить о реальной жизни, когда постоянным изменениям подвергаются тысячи функций из сотен библиотек и множество других видов внешних интерфейсов.

И здесь на помощь разработчикам приходят стандарты, определяющие, на какие интерфейсы в дистрибутиве можно полагаться, а какие подвергаются частым изменениям и стабильными не являются. На какие именно стандарты следует ориентироваться разработчику, зависит от того, какое приложение он создает.

Переносимость приложений, компилируемых в бинарные файлы

Существенная часть приложений для Linux разрабатывается на компилируемых языках (в основном C/C++). Переносимость таких приложений можно обеспечивать на двух уровнях:

на уровне бинарного кода – файлы приложения, скомпилированные в одной системе, можно без пересборки использовать в других дистрибутивах;

на уровне исходного кода – один и тот же код без модификаций собирается в каждой конкретной системе.

В обоих случаях для обеспечения переносимости необходимо, чтобы целевые дистрибутивы предоставляли библиотеки и функции, требуемые приложению.

Для совместимости на уровне исходных кодов необходимо также наличие заголовочных файлов со всеми необходимыми декларациями и прочих элементов API (Application Programming Interface, интерфейс программирования приложений). Унификация API различных ОС (не только Linux) является одной из основных целей стандарта POSIX (Portable Operating System Interface for Unix – переносимый интерфейс операционных систем UNIX). Как следует из расшифровки, изначально стандарт разрабатывался для различных вариаций UNIX. В настоящее время с POSIX совместимы большинство UNIX-подобных систем, включая дистрибутивы Linux.

Недостатком POSIX является его малый охват. Спецификации определяют всего около 1500 достаточно низкоуровневых системных интерфейсов – в частности, функции для работы с файлами, базовые математические операции, межпроцессорное взаимодействие и так далее. Ничего более высокоуровневого (например, библиотек графического интерфейса пользователя) стандарт не касается. Так что спектр приложений, которым достаточно только функций из POSIX, весьма узок.

Для обеспечения двоичной переносимости необходима совместимость целевых систем на уровне бинарного интерфейса приложений (Application Binary Interface, ABI).

В первую очередь требуется наличие всех необходимых функций и глобальных переменных в бинарных разделяемых библиотеках системы. Кроме того, необходимо использование одинаковых форматов бинарных файлов и разделяемых библиотек, одинаковых конвенций вызова функций и прочих незаметных, но важных аспектов.

При этом ряд ключевых составляющих ABI зависит от аппаратной платформы, поэтому переносимость на бинарном уровне возможна только в рамках одной аппаратной архитектуры (то есть бинарные файлы, скомпилированные для ARM, не запустятся на x86).

Разработчиков приложений под Linux, желающих добиться совместимости своих продуктов с большим количеством дистрибутивов, ждет немало серьезных препятствий

Однако в рамках одной архитектуры основное различие между дистрибутивами Linux заключается в наборе библиотек и их функций. Во всех прочих аспектах разные варианты этой ОС практически идентичны – для исполнимых файлов и разделяемых библиотек используется хорошо документированный формат ELF, конвенции вызова функций диктуются ставшим стандартом де-факто в Linux набором компиляторов GCC и так далее.

Именно перечень библиотечных функций, которые должны присутствовать в дистрибутиве, занимает существенную часть стандарта Linux Standard Base (LSB), нацеленного на обеспечение совместимости дистрибутивов на уровне ABI. Подход LSB заключается во включении в стандарт библиотек и функций, уже присутствующих во всех основных дистрибутивах и строго следящих за обратной совместимостью. При этом в первую очередь добавляются функции, наиболее востребованные приложениями, для чего производится постоянный мониторинг более чем тысячи современных программ.

LSB гораздо шире POSIX; последняя версия стандарта, LSB 4.1, включает почти 40 000 интерфейсов из 50 библиотек, затрагивая в том числе графический интерфейс пользователя, работу с мультимедиа и другие важные области. Впрочем, современные дистрибутивы содержат до нескольких тысяч библиотек и миллионы функций; на этом фоне LSB не кажется таким уж большим, и подавляющая часть библиотек остается вне его рамок.

Впрочем, LSB – динамично развивающийся стандарт. Новые версии выходят примерно раз в три года, и в каждой добавляется порядка нескольких тысяч функций. Начиная с версии 3.0, выпущенной в 2005 году, разработчики LSB придерживались обратной совместимости в рамках самого стандарта – приложения, совместимые с LSB 3.0, будут работать в системах, удовлетворяющих всем версиям LSB от 3.0 до 4.1 (выпущена в 2011 году) включительно. И только в LSB 5.0 планируется нарушить этот принцип, убрав из стандарта устаревшую библиотеку Qt3.

Отметим, что в большинстве случаев приложение, переносимое на уровне исходного кода, переносимо и на уровне бинарных файлов. Исключения происходят в ситуациях, когда одна и та же функция в одних системах реализована как макроопределение или встраиваемая (inline) функция, а в других – как обычная функция в библиотеке. Однако такие ситуации в Linux встречаются редко.

Наконец, необходимо отметить, что формально дистрибутивы Linux не проходят сертификацию на соответствие POSIX, а сертификаты LSB обычно получают только ОС, ориентированные на промышленное использование (например, Red Hat Enterprise Linux и SUSE Linux Enterprise), поскольку такой сертификат стоит некоторых денег.

Однако по факту разработчики базовых библиотек Linux следуют POSIX (и даже участвуют в его развитии), а создатели многих дистрибутивов проверяют свои системы на соответствие LSB – благо все тесты LSB находятся в открытом доступе.

В частности, при разработке ROSA регулярный запуск автоматических тестов LSB является частью процесса тестирования дистрибутива, так что мы уверены, что наша система удовлетворяет всем требованиям стандарта.

Переносимость приложения на интерпретируемых языках

Помимо традиционных C и C++, в Linux распространены и приложения на интерпретируемых языках и на языках, компилируемых в промежуточный байт-код, таких как Java, Perl, Python, Ruby и им подобные. Ситуация с переносимостью таких приложений обстоит лучше, чем в случае бинарных программ.

Причина этого в том, что интерпретаторы и основные библиотеки, покрывающие нужды большинства пользователей, развиваются в рамках единых проектов, строго следящих за обратной совместимостью. API базовых библиотек таких языков обычно изменяется в сторону добавления новых возможностей, а изменения, не совместимые с предыдущими версиями, допускаются крайне редко.

Например, для разработчиков Python есть формальный документ, предписывающий сохранять обратную совместимость (http://www.python.org/dev/peps/pep-0387), а Java от Oracle придерживается схожей политики уже много лет; некоторые нарушения между версиями JDK случаются, но они тщательно описываются.

Для многих языков существуют альтернативные реализации интерпретаторов и виртуальных машин для исполнения байт-кода, однако авторы таких реализаций стараются строго следовать спецификациям интерпретаторов и их библиотек – в частности, предоставлять все интерфейсы, имеющиеся в основной реализации, и следить за обратной совместимостью своих продуктов.

Популярность Perl и Python вкупе со стабильностью их основных библиотек послужила причиной добавления этих языков в LSB. Стандарт определяет минимальные версии интерпретаторов, которые должны присутствовать в каждом дистрибутиве, местоположение интерпретаторов в системе и набор базовых модулей.

Помимо стандартных библиотек и модулей, существует и множество сторонних дополнений и расширений – такие, как богатейшие коллекции модулей Perl на CPAN (Comprehensive Perl Archive Network, http://www.cpan.org) или модулей Python на PyPI (Python Package Index, https://pypi.python.org). Как правило, никаких стандартов и спецификаций на такие дополнения нет, и использующим их разработчикам остается надеяться на то, что создатели

дополнений следуют примеру основных библиотек и заботятся об обратной совместимости в своих релизах.

Установка приложения в ОС

Допустим, что вы собрали свое приложение, совместимое с основными представителями семейства Linux, и хотите дать пользователям ссылку на его скачивание, не дожидаясь, пока мэйнтейнеры дистрибутивов соберут его под свои системы. Однако что именно должен скачать пользователь, и какие действия предпринять, чтобы установить ваше приложение?

Наиболее предпочтительный способ – это добиться бинарной совместимости продукта с различными вариантами Linux и передавать пользователю набор скомпилированных файлов. Вам понадобятся отдельные сборки под каждую аппаратную архитектуру, но набор распространенных архитектур не очень широк – для настольных машин это x86 и x86-64, а для мобильных устройств – еще и различные вариации ARM.

В какой форме передать эти файлы пользователю? Самый простой вариант – просто запаковать их в архив (можно в самораспаковывающийся – чтобы пользователь не обращался к сторонним приложениям для извлечения файлов). Этот способ надежен, но не лишен недостатков. Первый из них – невозможность запросить у пользователя какие-то данные, которые могут понадобиться для установки, и выполнить какие-то действия, помимо распаковки, например, прописать приложение в меню запуска.

С такой проблемой борются посредством создания несложных программ установки. В первом приближении это может быть простой скрипт на языке оболочки Shell (такой подход часто используют в различных серверных продуктах, которые скорее всего будут устанавливаться администраторами, знакомыми с командной строкой). Можно сделать и более дружественный к обычным пользователям вариант, например, с использованием InstallAnywhere от создателей InstallShield, знакомого многим пользователям Windows, или свободного Autopackage.

Однако использование такого установщика не спасает от еще одной проблемы. Дело в том, что практически во всех дистрибутивах Linux используются централизованные системы управления ПО, с помощью которых осуществляется установка приложений из репозиториев. Для пользователя было бы удобнее, чтобы с помощью этой системы можно было бы устанавливать и удалять и сторонние приложения, но как этого достичь?

К сожалению, простого и универсального ответа на этот вопрос в настоящее время нет. Наиболее логичным является вариант упаковки файлов приложения в пакет, с которым способна работать система управления ПО дистрибутива. Подробнее о пакетах мы рассказывали в статьях о разработке дистрибутивов Linux в предыдущих номерах СА [1-3]. Здесь же напомним, что разные дистрибутивы используют разные форматы пакетов (наиболее популярными являются RPM и Deb), а также разные подходы к формированию зависимостей между пакетами. Эти различия приводят к тому, что собрать пакет, подходящий для всех дистрибутивов, очень сложно, а зачастую просто невозможно.

За время развития Linux было предпринято немало попыток предложить пути решения проблемы установки сторонних приложений. В частности, создатели стандарта LSB тоже не обошли ее вниманием, предложив использовать для распространения приложений пакеты в формате RPM версии 3. На сегодняшний день этот формат устарел, однако системы, основанные на RPM версий 4 и 5, а также утилита alien, конвертирующая пакеты RPM в Deb, способны устанавливать и пакеты RPM3.

Остается констатировать, что существующие способы распространения сторонних приложений для Linux далеки от идеала

Вопрос с зависимостями в LSB решается просто: если ваше приложение соответствует LSB, то вы просто указываете в качестве зависимости пакет lsb и при необходимости ограничения на версию стандарта, которой должен соответствовать дистрибутив, чтобы запустить ваше приложение. Все совместимые с LSB дистрибутивы имеют пакет с названием «lsb», при установке которого устанавливаются все программные компоненты, необходимые для работы LSB-совместимых приложений.

Как быть в случае, если ваше приложение не полностью удовлетворяет LSB и ему требуются компоненты, не устанавливаемые при установке пакета lsb, – стандарт умалчивает. А поскольку, несмотря на достаточно большой объем стандарта, многие приложения все-таки используют не входящие в него библиотеки и функции, то на практике предлагаемый в LSB подход почти не используется. Впрочем, о проблемах стандартов и о том, насколько они достигают своих целей, мы поговорим в следующей части статьи.

Пока что остается констатировать, что существующие способы распространения сторонних приложений для Linux далеки от идеала. За последнее десятилетие было несколько попыток «примирить» различные системы управления пакетами – в частности, уже упоминавшийся Autopackage, опирающийся на файловые зависимости, или разработанный не без помощи рабочей группы LSB набор функций Berlin Packaging API, работающий поверх стандартных инструментов управления пакетами. Однако на данный момент ни один из предлагавшихся подходов не получил сколько-нибудь заметного распространения. Единственным позитивным сдвигом стало внедрение автоматических генераторов зависимостей в инструментарий RPM – это позволяет гарантировать унифицированный подход к именованию по крайней мере части зависимостей в пакетах этого формата (например, современный инструментарий RPM автоматически добавляет зависимости от библиотек, используемых в бинарных файлах приложения, и при установке пакета автоматически ставятся все нужные библиотеки).

Как итог: большинство независимых производителей проприетарного ПО для Linux предлагают скачать архив с файлами приложения, либо пакет в формате RPM – типичным примером здесь является Java от Oracle. Встречаются и приложения с красивыми инсталляторами. Однако

проблема зависимостей, по большому счету, так и остается неразрешенной, и многие производители предпочитают паковать все необходимые библиотеки и прочие компоненты как часть приложения, нежели надеяться на их наличие в системе пользователя.

Интеграция приложения с ОС

Разобравшись, с горем пополам, с вопросом доставки приложения пользователю, вспомним о том, что приложение должно корректно «вписаться» в систему. Для начала файлы приложения необходимо разместить в ОС согласно ее правилам. В целом структура файловых систем в разных дистрибутивах единообразна и следует стандарту File System Hierarchy Standard (FHS) – http://refspecs.linuxfoundation.org/fhs.shtml с некоторыми добавлениями, описанными в LSB. В областях, незатронутых в FHS и LSB, дистрибутивы могут отличаться, но таких сфер немного.

Для размещения файлов приложений FHS определяет следующие основные директории:

/usr/bin – для исполняемых файлов приложений;

/usr/lib – для библиотек;

/usr/share – для файлов приложений, не зависящих от аппаратной архитектуры (в частности, сюда помещаются все ресурсы приложения – справочные материалы, иконки и прочее);

/opt – для крупных приложений, содержащих большое количество исполняемых файлов и библиотек.

Если ваша программа небольшая (например, представляет собой один исполняемый файл, набор иконок и справочное руководство), то вы можете разместить код в /usr/bin, а данные – в /usr/share, создав в ней поддиректорию с именем вашей программы. Библиотеки теоретически можно помещать в /usr/lib (на 64-битных системах – /usr/lib64), однако следует помнить, что в этой же директории лежат системные библиотеки. И если ваше приложение поставляется с библиотеками, которые могут дублировать системные, то ставить их в /usr/lib ни в коем случае нельзя, в этом случае желательно устанавливать все приложение в директорию /opt.

При установке в /opt внутри этой директории следует создать поддиректорию с именем приложения, внутри которой воссоздать структуру директории /usr, то есть сделать папку bin с запускаемыми файлами, lib с библиотеками и share с данными. В целом директория /opt по своему назначению напоминает Program Files в Windows, однако в отличие от последней в Linux специфицируется и структура поддиректорий для каждого приложения.

Важным аспектом при установке приложения в дистрибутиве является предотвращение конфликтов с другими программными компонентами – по файлам, библиотекам и так далее. При условии, что имя приложения уникально, а библиотеки и другие внешние интерфейсы приложения не регистрируются в качестве системных, установка всех файлов в директорию /opt/имя_приложения позволяет избежать большинства проблем. Тем не менее для полной гарантии уникальности каждому приложению необходимо иметь некоторый уникальный идентификатор, который можно использовать в именах файлов и интерфейсов.

Этот вопрос также удостоился внимания разработчиков стандарта LSB, который в качестве подобного уникального идентификатора предлагает использовать имя домена в Интернете, где располагается проект (подход, наверняка знакомый разработчикам на Java, где для именования классов также используются доменные имена). Однако доменное имя слишком длинное и не очень удобное. Для тех, кто хочет иметь более короткий идентификатор, предусмотрен единый реестр имен, поддерживаемый в рамках проекта LANANA (Linux Assigned Names And Numbers Authority, http://www.lanana.org/). Принцип работы реестра схож с распределением доменных имен в Интернете – вы подаете заявку на использование того или иного идентификатора, ее рассматривает соответствующая рабочая группа и при отсутствии конфликтов с существующими записями добавляет в реестр.

Завершая тему интеграции с ОС, нельзя не затронуть ряд спецификаций, разработанных в рамках проекта freedesktop.org (ранее называвшегося X Desktop Group, вследствие чего многие спецификации содержат в своем названии аббревиатуру «XDG»). Полный список спецификаций можно найти по адресу: http://www.freedesktop.org/wiki/Specifications. Не пугайтесь их количества – по объему большинство из них умещается в десяток страниц текста. Основной интерес для разработчиков приложений представляют документы, описывающие способы взаимодействия их продуктов с ОС, не зависящие от используемой графической среды.

В частности, на сайте можно найти описания добавления приложения в автозапуск и работы с конфигурационными файлами пользователя, а также спецификации desktop-файлов, содержащих общую информацию о приложении (имя, описание, путь к иконке, путь к запускаемому файлу и тому подобное, эти файлы используются при построении меню приложений и для привязки приложений к форматам файлов). При использовании спецификаций XDG учтите, что формально они являются не стандартами, а всего лишь рекомендациями. На сайте явно указано, какие документы уже стали стандартами де-факто, а какие поддерживаются только ограниченным числом реализаций.

***

Таким образом, разработчиков приложений под Linux, желающих добиться совместимости своих продуктов с большим количеством дистрибутивов, ждет немало серьезных препятствий. Существенным подспорьем в этом непростом деле являются стандарты и спецификации. Насколько легко применять эти стандарты в реальной жизни, какие проблемы разработчиков они действительно решают и какие трудности испытывают разработчики самих спецификаций, а также какие инструментальные средства поддержки стандартов существуют в мире Linux – обо всем этом речь пойдет в следующей части статьи. EOF

1. Силаков Д. Что такое дистрибутив Linux? Разработка дистрибутива Linux на примере РОСЫ. Часть 1. //«Системный администратор», №1-2, 2013 г. – С. 120-124.
2. Силаков Д. Что такое дистрибутив Linux? Разработка дистрибутива Linux на примере РОСЫ. Часть 2. //«Системный администратор», №3, 2013 г. – С. 83-87.
3. Силаков Д. Контроль качества дистрибутива Linux. //«Системный администратор», №4, 2013 г. – С. 82-86.