Визитка

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

Стандарты в Linux
Инструментальная поддержка и проблемы

В прошлый раз мы разобрались с тем, как происходит стандартизация в мире Linux, а теперь разберемся с техническими средствами и инструментами, которые в этом помогают

Современный стандарт в виде текста не всегда полезен и удобен – например, LSB в бумажной интерпретации занял бы несколько томов. А поскольку он ссылается более чем на сотню других стандартов и спецификаций (включая POSIX, OpenGL и другие достаточно объемные документы), то пользоваться им в бумажном варианте практически нереально. Наличие электронного представления текста в виде HTML или PDF несколько улучшает ситуацию, но не снимает проблему огромного объема информации.

Поэтому создатели многих стандартов предоставляют не только текст, но и различные вспомогательные инструментальные средства, призванные помочь в создании совместимых со стандартом продуктов.

Тесты

Одним из часто используемых вспомогательных средств является тестовый набор, проверяющий соответствие конкретной реализации требованиям стандарта.

В случае стандартов на интерфейсы взаимодействия приложений с ОС тесты могут быть нацелены на две различные категории продуктов:

- тесты для компонентов ОС, предоставляющих определенные в стандарте интерфейсы;
- тесты для приложений, использующих интерфейсы.

Задача тестов первого вида – удостовериться, что дистрибутив предоставляет все необходимые интерфейсы с требуемыми характеристиками.

Задача тестов второго вида – помочь разработчикам приложений убедиться, что их продукты используют только разрешенные стандартом интерфейсы и только таким образом, каким это предусмотрено стандартом.

Если стандарт подразумевает формальную процедуру сертификации продуктов на соответствие требованиям спецификации, то именно успешное прохождение тестов является основным (а зачастую и единственным) условием получения такого сертификата.

Рассмотрим реальный пример: LSB требует присутствия в системной библиотеке libc функции ioctl(), осуществляющей различные операции, связанные с вводом-выводом (ioctl расшифровывается как input/output control). В различных дистрибутивах Linux (в зависимости от версий ядра и библиотеки libc) функция ioctl() может поддерживать различные виды аргументов и соответственно осуществлять различные операции.

В стандарте LSB специфицировано только подмножество возможных аргументов ioctl(), которые должны гарантированно поддерживаться в любой LSB-совместимой ОС. Соответственно тесты LSB для дистрибутивов проверяют, что в системе присутствует библиотека libc, экспортирующая функцию ioctl(), которая принимает на вход все виды предусмотренных стандартом аргументов и выполняет соответствующие операции.

Тесты LSB для приложений пытаются проверить, что при вызове ioctl() приложение не передает функции аргументов, не предусмотренных стандартом.

Подчеркнем в последнем предложении слово «пытаются» – не так-то просто проверить, что приложение использует функцию корректно при любых сценариях работы.

Тестовый инструментарий LSB DynChk для приложений предоставляет средства для отслеживания аргументов, передаваемых ioctl() в процессе работы программы, но сам процесс запуска приложения в различных условиях отдается на откуп человеку, запускающему тесты.

Поэтому, если разработчики тщательно подходят к тестированию своего продукта и имеют набор автоматизированных тестов или тест-планов, то LSB DynChk можно интегрировать в процесс тестирования.

Эталонные реализации

Создатели некоторых стандартов предоставляют разработчикам так называемую эталонную реализацию (sample implementation) своих спецификаций. Такая реализация формально отвечает всем требованиям стандарта и даже может быть вполне работоспособной (хотя последнее не обязательно).

Эталонная реализация обычно служит двум целям:

- во-первых, разработчики реальных продуктов, которые должны предоставлять все интерфейсы стандарта, по-

Полноценные реализации

Разработчики проекта freedesktop.org пошли дальше эталонных реализаций и создали набор утилит xdg-utils, полноценно воплощающих в жизнь спецификации XDG. Основная цель этих инструментов – позволить разработчикам приложений выполнять ряд действий (добавлять и удалять иконки и пункты системного меню, работать с хранителем экрана, отправлять электронные письма или открывать файлы приложением, ассоциированным с ними в системе) независимым от конкретного дистрибутива образом.

Справедливости ради отметим, что спецификации XDG по сравнению с LSB или POSIX очень просты и охватывают узкий класс задач. Поэтому создание утилит, реализующих эти спецификации, не составляет большой сложности. Фактически xdg-utils являются обертками, определяющими, в какой графической среде (KDE, Gnome, Unity и так далее) они работают, и в зависимости от этого вызывающими соответствующие инструменты графического окружения.

лучают некий каркас, который они могут дорабатывать под свои нужды и исходя из своих интересов;

- во-вторых, программисты, чьи приложения используют интерфейсы из стандарта, получают полигон для тестирования своих продуктов.

Примером эталонной реализации в мире открытых стандартов является OpenGL Sample Implementation от SGI, на основе которой было создано немало драйверов для видеокарт (во всяком случае, тех их частей, которые предоставляют OpenGL API).

Из рассмотренных в первой части статьи [1] стандартов попытку создать эталонную реализацию предпринимали разработчики LSB, предлагавшие небольшой дистрибутив Linux (на основе Linux From Scratch), предоставляющий все интерфейсы, определяемые стандартом, и ничего лишнего. Однако оказалось, что на практике поддержка такой реализации LSB слишком сложна и затратна.

К тому же LSB Sample Implementation не пользовалась большой популярностью ни у разработчиков приложений (предпочитающих тестировать свои продукты в реальных дистрибутивах), ни у создателей дистрибутивов (которым никакой каркас для создания LSB-совместимой системы не нужен – технически гораздо проще доработать уже существующую реализацию). Поэтому разработка LSB Sample Implementation была свернута несколько лет назад.

Инструментарий разработчика

Помимо различных средств для проверки соответствия программного продукта стандарту, разработчикам могут предоставляться и средства для непосредственного создания такого продукта. Примерами таких инструментов являются специализированные наборы средств разработки ПО (Software Development Kit, SDK).

В частности, один такой набор – LSB SDK – развивается рабочей группой LSB. LSB SDK предназначен для сборки приложений, написанных на языках C и C++, и содержит следующие компоненты:

- заголовочные файлы, определяющие только функции, входящие в LSB, необходимые для использования этих функций типы данных, а также макроопределения и константы, использование которых не противоречит LSB;
- библиотеки-заглушки, экспортирующие только функции, определенные в стандарте; эти заглушки не содержат реализаций функций и используются только на этапе компоновки приложения, и если программа попытается использовать какую-то функцию не из LSB, то ее компоновка с библиотеками-заглушками завершится неудачно;
- обертки для системных компиляторов, при вызове которых вместо системных заголовочных файлов и библиотек используются файлы и библиотеки-заглушки из LSB SDK.

Обертки для компиляторов также выставляют системным компиляторам ряд опций, в отсутствие которых вы можете получить несовместимое с LSB приложение, даже не используя явно неразрешенных функций.

Использование LSB SDK дает очень высокую гарантию того, что полученное в результате сборки приложение соответствует LSB. Именно специализированный SDK стал одной из причин низкого интереса к эталонной реализации LSB со стороны разработчиков приложений, нацеленных на соответствие этому стандарту, – оказалось, что проще собрать приложение с помощью LSB SDK и протестировать его в одном из дистрибутивов, соответствующих LSB, чем использовать отдельную ОС специально для тестов.

Инструменты повышения переносимости приложений

Рассказав об инструментах, нацеленных на обеспечение совместимости приложения с некоторым стандартом, нельзя обойти стороной ряд средств, предназначенных для увеличения переносимости ПО без привязки к каким-либо стандартам.

Наиболее известным в мире Linux примером инструментария, призванного повысить переносимость программ на уровне исходного кода, являются утилиты GNU Autotools. Этот набор средств позволяет программистам абстрагироваться, например, от таких особенностей конкретной системы, как местоположение тех или иных заголовочных файлов и библиотек.

Autotools часто используют в сочетании с инструментом pkg-config, позволяющим автоматически получить параметры компилятора и компоновщика, необходимые для сборки приложения с той или иной библиотекой или стеком библиотек.

Autotools нередко подвергались критике за излишне сложную структуру (см. рис. 1), и в последние годы появились достойные альтернативы этому инструментарию, уже получившие широкое распространение, – CMake и Scons.

Что касается переносимости ПО на бинарном уровне, то здесь хорошим подспорьем является Linux Application Checker от Linux Foundation.

Этот инструмент разрабатывался членами рабочей группы LSB и изначально содержал утилиты проверки соответствия приложений этому стандарту.

Однако в настоящее время, помимо данных утилит Application Checker, предоставляет возможность проанализировать, в каких дистрибутивах может быть запущено приложение.

Анализ проводится путем сравнения набора библиотек и функций, требующихся бинарным файлам приложения,

с библиотеками и функциями, предоставляемыми различными дистрибутивами.

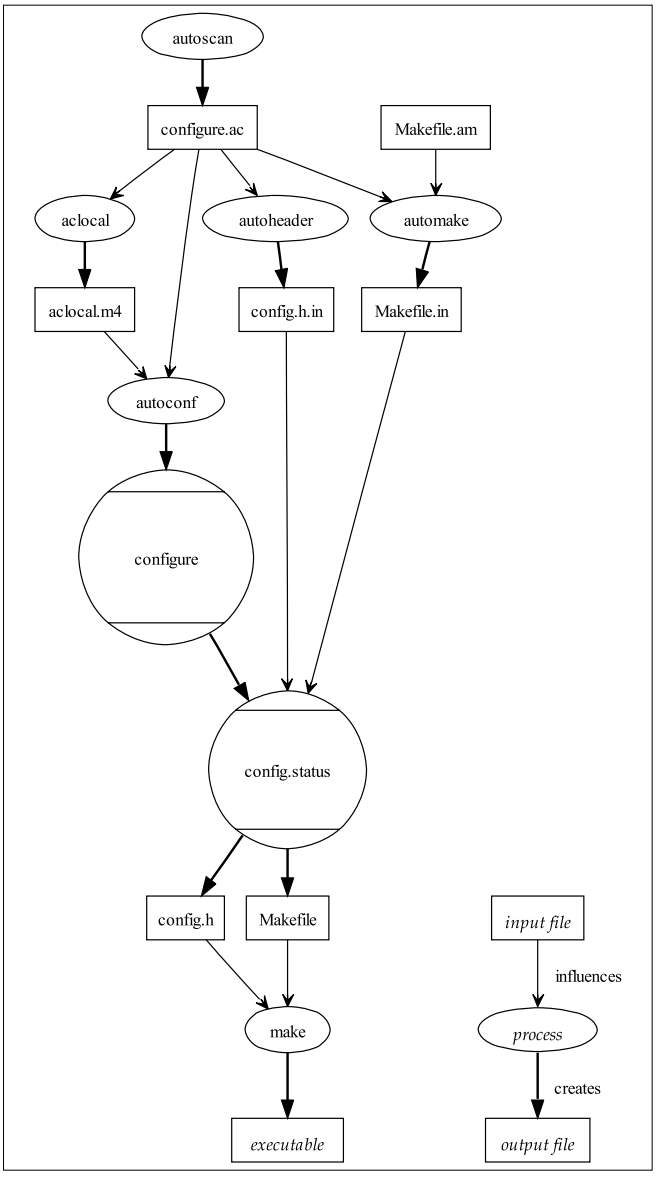
На данный момент Application Checker содержит информацию почти о 90 различных версиях дистрибутивов, среди которых Red Hat Enterprise Linux, SUSE Linux Enterprise Server, Ubuntu, Debian и другие системы. Отечественные разработки представлены ROSA 2012 Marathon.

Инструменты и сервисы для разработчиков дистрибутивов

Наряду с инструментарием, помогающим разработчикам приложений, существуют различные утилиты и сервисы для создателей компонентов дистрибутивов, упрощающие задачу поддержки обратной совместимости этих продуктов.

Одним из таких инструментов является ABI Compliance Checker (ACC) [1], созданный при поддержке Linux Foundation и в настоящее время развиваемый разработчиками РОСЫ.

Рисунок 1. Схема сборки программного проекта при использовании GNU Autotools – путь к результату не слишком короткий



ACC предназначен для сравнения совместимости двух различных версий библиотеки, написанной на C или C++. Инструментарий анализирует как заголовочные, так и двоичные файлы в формате ELF и проверяет различные структурные аспекты совместимости как на бинарном уровне, так и на уровне исходного кода. В частности, отслеживается удаление функций, изменение их параметров, модификации состава классов и сложных типов (структур, перечислений) и многое другое.

ABI Compliance Checker лежит в основе сервиса Upstream Tracker (http://upstream-tracker.org), предоставляющего информацию о совместимости различных версий более чем пятисот библиотек. Сервис осуществляет мониторинг сайтов разработчиков, на которых публикуются библиотеки, и при появлении новой версии автоматически проводит анализ ее совместимости с предыдущей. На рис. 2 и 3 приведена статистика Upstream Tracker для библиотек OpenAL (почти образец стабильности) и OpenSSH (здесь разработчики любят внести немного несовместимых изменений в новый релиз).

Upstream Tracker пользуется популярностью у мэйнтейнеров дистрибутивов, позволяя им бегло оценить риски перехода на новую версию библиотеки – если обратная совместимость нарушена, то придется обновлять и пересобирать все зависящие от библиотеки приложения. Если же новая версия обратно-совместима с предыдущей, то переход должен пройти безболезненно.

Конечно, Upstream Tracker проверяет не все аспекты совместимости, и, если что-то радикально изменилось в поведении или алгоритме работы некоторой функции, это останется незаметным для анализатора. Но даже неполный анализ, проводимый сервисом, доказал свою полезность и уже сэкономил немало времени разработчикам.

Технические способы сохранения обратной совместимости

Различные тесты и прочие описанные выше инструменты позволяют обнаруживать нарушение обратной совместимости. Но как быть, если разработчикам все-таки необходимо внести какие-то изменения в библиотеку и при этом нет желания нарушить работоспособность зависящих от нее программ?

Оказывается, системные средства Linux предоставляют достаточно гибкие возможности по безболезненному осуществлению подобных изменений.

Начнем с того, что библиотекам в Linux (равно как и во многих других ОС) можно явно приписывать версию. Имя библиотеки вместе с основной версией (например, libc.so.6) является уникальным идентификатором (так называемым soname), по которому с ней работает динамический загрузчик, определяющий, какие библиотеки нужно загрузить в память для работы той или иной программы. Соответственно в бинарные файлы программ прописываются зависимости именно от soname.

При этом вовсе не обязательно помещать в soname полную версию библиотеки. Традиционно в Linux версии многих продуктов состоят как минимум из двух частей – мажорной и минорной, разделенных точкой, – например, libjpeg-8.1. Часто еще через одну точку добавляют микроверсию – libgtk-2.24.1.

В soname при этом включают только мажорную часть версии, которую увеличивают, только если в библиотеке происходят изменения, нарушающие обратную совместимость. Поэтому релизы, сохраняющие обратную совместимость, имеют одинаковую мажорную версию – это правило справедливо для многих библиотек в Linux.

Традиционно версия отражается в имени файла библиотеки – например, libXtst.so.6.1.0. Если имя библиотеки не совпадает с ее soname, то обязательно создается символьная ссылка с именем, совпадающим с soname, указывающая на реальный файл.

Такой набор файлов библиотек и ссылок на них можно наблюдать в директории /usr/lib (или /usr/lib64 в 64-битных системах), в которую устанавливается большинство системных библиотек.

Если количество изменений, нарушающих обратную совместимость, невелико, то можно использовать другой подход – версионирование бинарных символов. Этот подход подразумевает приписывание версии каждому символу (функции или глобальной переменной), экспортируемому библиотекой. Если в функцию вносится модификация, несовместимая с предыдущей версией, то код старой функции сохраняется под другим именем и указывается, что это имя соответствует функции со старой версией.

Для ясности рассмотрим, как это выглядит в коде. Пусть у нас есть функция func(), возвращающая единицу. Припишем этой функции версию VER_1:

```
void func() { return 1; }
__asm__(".symver func, func@VER_1");
```

(здесь мы для краткости воспользовались ассемблерной вставкой; вместо этого можно передать компоновщику файл с указанием, какому символу какую версию приписать).

Допустим, мы решили изменить функцию func(), чтобы она возвращала двойку. Для обеспечения обратной совместимости мы сохраним старую реализацию под именем func_old() и укажем, что библиотека должна экспортировать символ func с версией VER_1, который соответствует func_old. Новой же реализации мы припишем новую версию – VER_2.

```
void func() { return 2; }
__asm__(".symver func, func@VER_2");
void func_old() { return 1; }
__asm__(".symver func_old, func@VER_1");
```

При сборке приложений, использующих библиотеку, в них прописываются зависимости от конкретных версий символов. Поэтому если в коде приложения вызывается func() и оно собирается со старой версией библиотеки, то в зависимости от приложения будет прописан символ func@VER_1, а при использовании новой версии – func@VER_2. Новая версия библиотеки экспортирует оба символа, поэтому с ней смогут работать и приложения, собранные с использованием старой версии. А это и есть обратная совместимость.

Теоретически версионирование символов позволяет сохранять обратную совместимость сколь угодно долго. Но на практике оно приводит к увеличению размера библиотек (как исходного кода, так и бинарных файлов), ведь приходится «тащить» весь старый код. Причем старые реализации надо не просто переносить из одной версии в другую, но и поддерживать их в рабочем состоянии (поскольку они могут обращаться к другим функциям, реализации которых также изменяются) и исправлять ошибки и уязвимости.

Поэтому такой способ сохранения обратной совместимости используется далеко не всеми разработчиками и только в ситуациях, когда изменения относительно невелики и затрагивают небольшое количество функций. В противном случае проще выбросить старый код, изменить soname библиотеки и рекомендовать пересобрать все зависящие от нее приложения.

Вместо заключения – проблемы современных стандартов

Итак, несмотря на богатое многообразие вариаций Linux, существуют стандарты и спецификации, которым следуют практически все дистрибутивы. Также существуют различ-

Рисунок 2. Обратная совместимость релизов библиотек OpenAL

Version	Date	SONAME VERSION	Change Log	Backward Compatibility	Added Symbols	Removed Symbols	Headers Diff	API Changes[7] / Binary Compatibility		
								High	Medium	Low
4.9.6	2013-03-14	4	changelog	100%	1 new	0	diff	0	0	0
4.9.5	2013-01-31	4	changelog	100%	0	0	diff	0	0	1 warning
4.9.4	2012-11-21	4	changelog	100%	0	0	diff	0	0	1 warning
4.9.3	2012-10-23	4	changelog	100%	0	0	diff	0	0	1 warning
4.9.2	2012-07-20	4	changelog	100%	2 new	0	diff	0	0	0
4.9.1	2012-06-01	4	changelog	100%	0	0	diff	0	0	0
4.9	2012-02-16	4	changelog	100%	0	0	diff	0	0	0
4.8.9	2011-08-09	4	changelog	100%	1 new	0	diff	0	0	0
4.8.8	2011-05-06	4	changelog	100%	0	0	empty	0	0	0
4.8.7	2010-12-18	4	changelog	100%	0	0	diff	0	0	0
4.8.6	2010-07-09	4	changelog	100%	1 new	0	diff	0	0	0
4.8.4	2010-02-08	4	changelog	100%	0	0	diff	0	0	0
4.8.3	2009-12-15	4	changelog	100%	0	0	diff	0	0	0
4.8.2	2009-10-06	4	changelog	100%	0	0	diff	0	0	0
4.8	2009-05-13	4	changelog	100%	2 new	0	diff	0	0	0
4.7.6	2009-10-08	4	changelog	100%	0	0	diff	0	0	0
4.7.5	2009-06-13	4	changelog	100%	0	0	diff	0	0	0
4.7.4	2009-03-25	4	changelog	100%	0	0	diff	0	0	0
4.7.3	2008-11-04	4	changelog	100%	0	0	diff	0	0	0
4.7.2	2008-08-30	4	changelog	100%	0	0	diff	0	0	0
4.7.1	2008-04-29	4	changelog	100%	0	0	diff	0	0	0
4.7	2008-01-24	4	changelog	100%	2 new	0	diff	0	0	16 warnings
4.6.8	2007-10-31	4	changelog	100%	0	0	diff	0	0	0
4.6.7	2007-04-28	4	changelog	100%	0	0	diff	0	0	0
4.6.6	2007-01-19	4	changelog	100%	0	0	diff	0	0	0
4.6.5	2007-01-17	4	changelog	100%	0	0	diff	0	0	0
4.6.4	2006-11-14	4	changelog	100%	0	0	diff	0	0	0
4.6.3	2006-09-09	4	changelog	100%	0	0	diff	0	0	0
4.6.2	2006-04-28	4	changelog	100%	2 new	0	diff	0	0	1 warning
4.6.1	2005-09-02	4	changelog	100%	0	0	diff	0	0	0

Рисунок 3. Обратная совместимость релизов библиотек OpenSSH

Version	Date	SONAME VERSION	Change Log	Backward Compatibility	Added Symbols	Removed Symbols	Headers Diff	API Changes[7] / Binary Compatibility		
								High	Medium	Low
6.2p1	2013-03-22	n/a	changelog	97.2%	22 new	3 removed	diff	0	7 changes	11 warnings
6.1p1	2012-08-29	n/a	changelog	98.1%	1 new	0	diff	1 change	1 change	2 warnings
6.0p1	2012-04-20	n/a	changelog	98%	3 new	1 removed	diff	0	1 change	1 warning
5.9p1	2011-09-07	n/a	changelog	99.3%	5 new	2 removed	diff	0	1 change	0
5.8p2	2011-05-03	n/a	changelog	100%	0	0	diff	0	0	0
5.8p1	2011-02-04	n/a	changelog	100%	0	0	diff	0	0	0
5.7p1	2011-01-22	n/a	changelog	93%	29 new	3 removed	diff	0	5 changes	3 warnings
5.6p1	2010-08-23	n/a	changelog	94.8%	4 new	0	diff	1 change	4 changes	6 warnings
5.5p1	2010-04-16	n/a	changelog	100%	1 new	0	diff	0	0	0
5.4p1	2010-03-08	n/a	changelog	97.1%	18 new	1 removed	diff	1 change	8 changes	11 warnings
5.3p1	2009-09-26	n/a	changelog	99%	19 new	2 removed	diff	14 changes	2 changes	0
5.2p1	2009-02-23	n/a	changelog	95.9%	4 new	0	diff	1 change	2 changes	1 warning
5.1p1	2008-07-21	n/a	changelog	94.9%	13 new	1 removed	diff	1 change	11 changes	8 warnings
5.0p1	2008-04-03	n/a	changelog	100%	0	0	diff	0	0	0
4.9p1	2008-03-27	n/a	changelog	100%	2 new	0	diff	0	0	0
4.7p1	2007-09-04	n/a	changelog	97.5%	6 new	0	diff	1 change	2 changes	8 warnings
4.6p1	2007-03-06	n/a	changelog	100%	0	0	diff	0	0	0
4.5p1	2006-11-07	n/a	changelog	100%	0	0	diff	0	0	0
4.4p1	2006-09-26	n/a	changelog	96.9%	16 new	0	diff	1 change	12 changes	6 warnings
4.3p2	2006-02-11	n/a	changelog	100%	0	0	diff	0	0	0
4.3p1	2006-02-01	n/a	changelog	93.9%	6 new	0	diff	4 changes	10 changes	4 warnings
4.2p1	2005-09-01	n/a	changelog	99.1%	4 new	1 removed	diff	0	2 changes	6 warnings
4.1p1	2005-05-25	n/a	changelog	100%	0	0	diff	0	0	0
4.0p1	2005-03-09	n/a	changelog	95.4%	17 new	0	diff	5 changes	2 changes	4 warnings
3.9p1	2004-08-17	n/a	changelog	94.3%	10 new	0	diff	0	4 changes	6 warnings
3.8.1p1	2004-04-18	n/a	changelog	100%	0	0	diff	0	0	0
3.8p1	2004-02-24	n/a	changelog	98.4%	6 new	5 removed	diff	0	0	39 warnings
3.7.1p2	2003-09-23	n/a	changelog	100%	0	0	diff	0	0	0
3.7.1p1	2003-09-17	n/a	changelog	100%	0	0	diff	0	0	0
3.7p1	2003-09-16	n/a	changelog	83.5%	19 new	43 removed	diff	0	4 changes	11 warnings
3.4p1	2002-06-26	n/a	changelog	99.7%	0	0	diff	0	1 change	8 warnings
3.3p1	2002-06-21	n/a	changelog	99.1%	6 new	3 removed	diff	0	0	2 warnings
3.2.3p1	2002-05-22	n/a	changelog	100%	0	0	diff	0	0	0
3.2.2p1	2002-05-16	n/a	changelog	99.7%	50 new	0	diff	0	1 change	2 warnings

ные технологии и инструментальные средства, позволяющие, с одной стороны, повысить совместимость различных вариантов свободной ОС, а с другой, добиться совместимости приложений с существенной частью таких систем.

Тем не менее со стороны разработчиков регулярно раздаются жалобы на сложность создания продуктов, пригодных для использования в любом дистрибутиве без каких-либо изменений. В частности, этими сложностями нередко объясняют отсутствие большого количества крупных проприетарных приложений. Нетрудно сделать вывод, что стандартизация охватила далеко не все области в мире Linux. В чем же причины такой ситуации?

Во-первых, как уже отмечалось ранее, количество всевозможных интерфейсов, которые могут быть задействованы для взаимодействия приложений с ОС, чрезвычайно велико, и охватить их все каким-то стандартом не представляется возможным. 40 000 функций в LSB – это, конечно, хорошо, но в дистрибутивах-то их миллионы.

Во-вторых, мир Linux изменяется очень быстро, и постоянно появляются новые технологии и наработки. При этом разные группы разработчиков представляют разные способы решения схожих проблем и не всегда заботятся о совместимости своих подходов и об обратной совместимости с уже использующимися решениями, что ведет к размежеванию дистрибутивов. Так что проблематичность стандартизации многих сфер в свободном ПО – это следствие постоянной тяги к инновациям.

При этом стоит отметить, что далеко не всегда инициативы по стандартизации находят отклик у разработчиков дистрибутивов. Тот же LSB успешно справляется с закреплением вещей, ставших стандартом «де-факто», но почти не предлагает новых решений, которые бы были приняты дистрибутивами.

На сайте проекта freedesktop.org представлено несколько десятков спецификаций [2], но только 11 из них помечены как документы, которым реально следует большинство реализаций. В два раза больше спецификаций числятся в состоянии «новые и еще не получившие широкого распространения» (хотя некоторые из них были написаны еще десять лет назад и скорее всего перейдут из состояния «новые» сразу в «устаревшие/неактуальные»).

И еще почти столько же находится в стадии планирования и сбора требований (и также пребывает в этой стадии по несколько лет). Возможно, причина здесь в том, что основные предложения по стандартизации редко привносят какие-то революционные идеи, за реализацию которых с удовольствием бы взялись добровольцы и энтузиасты. Зато их внедрение нередко требует серьезной переработки существующих продуктов и подразумевает большой объем рутинного труда.

В-третьих, следует понимать, что разработка стандарта – достаточно хлопотное занятие. Включение интерфейса в какой-либо стандарт не сводится к добавлению его упоминания в тексте спецификации – необходимо соответствующим образом обновить артефакты, связанные со стандартом, – например, тесты. Ведь мало декларировать наличие некоторой функции во всех дистрибутивах; необходимо убедиться, что эта функция везде работает одинаково.

Однако с тестированием обратной совместимости у многих проектов дела обстоят не очень хорошо. Несмотря на наличие различного рода автоматизированных средств, соблюдение требований обратной совместимости и отслеживание регрессий требует серьезных человеческих ресурсов. Как следствие, даже для основных библиотек и утилит, входящих в стандарты, не всегда есть качественные тесты, и регрессии иногда случаются. Качество же многих продуктов, развиваемых энтузиастами, оставляет желать лучшего, и об их стандартизации речи идти не может.

Таким образом, многообразие дистрибутивов Linux по-прежнему представляет проблему для разработчиков приложений. Степень схожести многих вариантов свободной ОС достаточно велика, однако гарантированно идентичными являются только области, охваченные различными стандартами и спецификациями, а многие компоненты дистрибутивов в эту категорию не попадают. Поэтому разработчики приложений для Linux, не рассчитывающие на помощь мэйнтейнеров, оказываются серьезно ограничены в плане технологий, которые они могут использовать.

В настоящее время среди производителей проприетарных приложений активно используется два подхода к решению этой проблемы – часть из них включает в состав своих продуктов множество сторонних компонентов, в наличии которых в системе они не уверены, другая часть просто ограничивается поддержкой только нескольких наиболее популярных дистрибутивов Linux. Ни один из этих подходов нельзя назвать удовлетворительным с точки зрения пользователей.

Однако Linux уже достиг зрелого возраста, и многие разработчики все чаще задумываются о вопросах стандартизации и совместимости, и есть все основания полагать, что в будущем ситуация с совместимостью дистрибутивов улучшится, а разработка переносимых приложений упростится. BOF

1. Силаков Д. Стандартизация в мире Linux. Зачем и как? //«Системный администратор», №5, 2013 г. – С. 82-86.
2. ABI Compliance Checker (ACC) – http://ispras.linuxbase.org/index.php/ABI_compliance_checker.
3. Спецификации проекта freedesktop.org – http://www.freedesktop.org/wiki/Specifications.