Построение встраиваемой ОС с использованием статического внедрения зависимостей

Существует важное отличие между ОС общего назначения и встраиваемыми ОС, несмотря на то , что они решают сходные задачи. ОС общего назначения обычно используется как платформа для разнообразных прикладных программ. С точки зрения разработчика такой ОС важно поддерживать неизменный набор предоставляемых функций и сервисов, с целью добиться совместимости со всеми существующими приложениями, а также совместимости версий ОС между собой. Напротив, встраиваемая ОС часто используется как конфигурируемый каркас для приложения (или приложений), используемых в данной конкретной встраиваемой системе и не должна содержать ничего лишнего. Кроме того, к ней гораздо более мягкие требования по части совместимости. Иными словами, если в мире прикладного ПО для персональных ПК приложения обычно пишутся «для ОС», то в вире встраивамого ПО дело обстоит с точностью до наоборот: ОС должна быть сконфигурирована под конкретное приложение, поэтому вопрос конфигурируемости особенно актуален именно для встраиваемых ОС. Поскольку в настоящее время С является основным языком для системного программирования, на конфигурируемость последних часто оказывают влияние ограничения, связанные с языком. В результате, в настоящее время существует множество операционных систем, область применения которых ограничена определенным классом встраиваемых систем (для контроллеров, для индустриальных компьютеров, для однопроцессорных / многопроцессорных систем и т.д.), хотя механизмы и алгоритмы, лежащие в основе ОС идентичны как для «маленьких» так и для «больших» ОС. В данной работе исследуется возможность использования внедрения зависимостей, для улучшения возможностей конфигурирования ОС и частичного устранения ограничений, накладываемых языком С в контексте разработки встраиваемых операционных систем.

Введение

История операционных систем началась в 60-70-х годах с разработки UNIX, ядро которой было монолитным, его компоненты были жестко связаны друг с другом. Параллельно с этим, проводились исследования в области разделения функциональности ядер, первым таким ядром была исследовательская разработка THE, разрабатываемая под руководством Э.Дейкстры. Исследования в области архитектуры ОС привели в настоящее время к концепции микроядер, где привилегиями супервизора обладает только небольшая часть ОС (микроядро), а все остальные сервисы ОС предоставляются процессами (такими же, как и пользовательские), микроядро же нужно, в основном, для связи процессов между собой (то есть для IPC).
Набор сервисов, требуемых от операционной системы общего назначения вполне четко определен, поэтому длительное время не возникало необходимости в конфигурировании ядер. Ситуация изменилась после миниатюризации компьютеров и появления встраиваемых систем, для которых на первое место вышли требования к ресурсам и конфигурируемость, так как в отличие от персональных компьютеров встроенные системы обладают в разы большим разбросом производительности процессора, объемов памяти и так далее.
Надо отметить что микроядра не предназначены для решения проблем конфигурируемости. Основная их цель – обеспечить защиту от ошибок в системных сервисах путем их выполнения с привилегиями пользователя. Достигаемая при этом «конфигурируемость», возможность запуска разных процессов, предоставляющих разные сервисы, может использоваться только во время выполнения и несет в себе дополнительные накладные расходы, связанные с косвенным общением между процессами, и поэтому плохо подходит для встраиваемых систем.
Многие виды функциональности, например поддержка процессов или виртуальной памяти имеет аспектный характер (влияет на многие модули в системе, не будучи связанной с ними отношениями наследования или включения). Поскольку язык C изначально создававшийся для разработки UNIX и до сих пор не сдающий позиций в системном программировании не имеет никаких средств для аспектно-ориентированного программирования, он плохо подходит для создания

конфигурируемых исходных текстов, это привело к написанию большого количества встраиваемых ОС, предназначенных для определенного круга целевых систем. Хотя все ОС решают сходные задачи и имеют сходные алгоритмы работы, код модулей крайне сложно использовать повторно, так как класс целевой системы уже определяет, будет ли определенный аспект присутствовать в реализации модуля или нет.

Модель использования встраиваемых ОС

В отличие от ОС общего назначения, которые имеют так называемую оболочку, с помощью которой приложения могут быть динамически загружены и выполнены, от встраиваемых ОС обычно не требуется такой функциональности. Многие из них жестко связываются с выполняемым приложением на этапе создания образа ОС. В случае если ОС не поддерживает разделение привилегий на ядро и пользовательские приложения, она может быть реализована просто как статическая библиотека, которая компонуется с приложением.

Метод конфигурирования с помощью внедрения зависимостей

Традиционными методами конфигурирования ОС является использование условной компиляции и настройки системы сборки на включение только определенных файлов в конечный образ. В первом случае конфигуриромость является «закрытой» - изменять можно только те параметры, и выбрать только те варианты, которые были учтены при написании кода. Во втором случае – при необходимости отключить определенный функционал необходимо использовать специальную бибилиотеку-заглушку, которая приводит к накладным расходам времени выполнения, что неприемлемо для встраиваемых систем.
Наилучшим образом для конфигурирования подходит так называемое внедрение зависимостей (ВЗ). Модули пишутся таким образом, что они используют некоторую функциональность на основе абстрактных интерфейсов. Конкретные реализации определяются на основе внешней метаинформации без изменения

исходных текстов.
Такие вещи как абстракция оборудования, включение/выключение логирования и трассировки и так далее естественным образом получаются в результате использования внедрения зависимостей. Плюсы использования ВЗ перечислены далее в этом разделе.

Открытость

Под открытостью понимается отсутствие ограничений на добавление сторонних модулей. То есть ОС не является «черным ящиком» в котором все возможности и конфигурационные параметры были заданы разработчиками при написании кода, а допускает неограниченное конфигурирование с помощью внешних модулей.
Все что нужно для использования нового модуля в ОС – скопировать его в папку, где он будет доступен для инструмента, ищущего интерфейсы, после этого он может быть внедрен в систему с помощью изменения метаинформации о зависимостях в системе.

Стандартный инструментарий

В отличие от аспектно-ориентированных языков, внедрение зависимостей может быть сделано в рамках стандартного синтаксиса языка C и C++, то есть внешние инструменты нужны только для упрощения рутинных операций, которые могут быть сделаны и вручную. При этом не требуется изучать аспектный синтаксис.

Широкий охват классов систем

Одновременно с расширением возможностей конфигурирования, ОС может применяться на большем количестве целевых устройств, так как функционал ОС может варьироваться в гораздо более широких пределах, чем в случае простого конфигурирования с помощью директив условной компиляции или системы сборки.
Это может оказаться существенным с точки зрения снижения

расходов, так как при разработке ПО сразу для нескольких классов устройств не придется использовать разные ОС, и, как следствие, разрабатывать слои абстракции для приложения.

Повторное использование кода

Поскольку используемые компоненты не требуют изменения кода и могут быть легко использованы в любом проекте, одни и те же компоненты могут использоваться при построении различных по свойствам ОС и даже для прикладных программ.

Простота

Поскольку каждый модуль представляет собой самостояельную сущность, которая может быть протестирована отдельно от всей остальной системы, сложность системы в целом снижается.

Упрощение тестирования

Так как модуль использует входные интерфейсы только абстрактно, упрощается юнит-тестирование - каждый модуль может быть помещен в искуственно создаенное окружение, моделирующее поведение любой реально возникающей ситуации, при этом даже не требуется сборка образа ОС, каждый компонент может тестироваться независимо от других.

Возможность независимой и параллельной разработки

Так как разработка модулей не требует от программиста знания всей системы, а требует только четкой спецификации импортируемых и экспортируемых интерфейсов, части ОС могут разрабатываться независимо друг от друга независимыми группами разработчиков.

Легкость конфигурирования

Поскольку конфигурирование осуществляется только с помощью изменения метаинформации, не требуется разбираться в структуре и организации файлов исходных текстов, конфигурации систем могут распространяться в виде отдельных файлов, не связанных с исходными текстами.

Возможность автоматического конфигурирования

В некоторых ситуациях возможно также автоматическое конфигурирование под нужды конкретного приложения без участия пользователя в этом процессе.

Общая архитектура

В первом приближении, поскольку отсутствуют какие-либо связи между компонентами до выполнения, собственно, внедрения зависимостей, изначально ОС представляет собой просто набор программных модулей, все они равноправны. Если модули имеют несколько реализаций, тогда в этом наборе компонентов содержится не одна ОС, а все возможные ОС, которые можно получить из этих компонентов с помощью конфигурирования.

Встраиваемая ОС FX-RTOS

Для воплощения вышеописанной концепции была разработана экспериментальная ОС FX-RTOS. Она представляет собой набор компонентов, связи между которыми определяются в результате работы механизма внедрения зависимостей до компиляции. Получаемая при этом система связывается полностью статически и не содержит накладных расходов, так как для каждого модуля может быть выбрана оптимальная реализация, а отсключаемые модули исключаются из исходных текстов (то есть не содержат накладных расходов времени выполнения в виде вызова функций-заглушек).
Из-за этого факта, FX-RTOS, строго говоря, не является ОС, так как не имеет четко определенной архитектуры, а является, скорее, набором повторно используемых компонентов, которые могут

быть использованы для построения самых различных ОС (и не только ОС). Тем не менее, для всех ОС построенных с помощью данных компонентов, характерны некоторые общие моменты, они будут рассмотрены далее.

Частный случай: Портирование

Поскольку функции платформы обычно используются абстрактно (вызываются функции, которые имеют одинаковую семантику для всех платформ и должны быть реализованы в слое абстракции оборудования), поддержка многих платформ осуществляется естественным образом с помощью внедрения зависимостей: в качестве реализаций абстрактных интерфейсов платформы в метаинформации указываются их реализации специфичные для конкретной аппаратной платформы.

Частный случай: Поддержка SMP

Важно отметить, что отключение поддержки SMP происходит на уровне исходного текста, а не на уровне компоновки модулей, как могло бы быть, в случае использования библиотек. Например функция get_current_cpu (возвращающая индекс текущего процессора), не будет заменена на функцию-заглушку, которая всегда возвращает 0, а будет заменена на макрос, который подставит само значение везде, где используется эта функция, то есть вызова функции не будет происходить, накладные расходы для однопроцессорных систем полностью отсутствуют, так как все места в коде наподобие:

```
If(get_current_cpu() == 0)
{
…
}
```

будут оптимизированы компилятором. В многопроцессорных системах будет вызываться реально существующая функция и выполняться сравнение, в однопроцессорных системах условие превратится в 0 == 0 (всегда истинно) и будет удалено из кода.

Частный случай: Поддержка нескольких

алгоритмов планирования

Для многих встраиваемых ОСРВ характерна поддержка нескольких алгоритмов планирования, которые могут меняться при конфигурировании. Эта проблема может быть решена по аналогии с проблемой портирования – поскольку интерфейс планировщика фиксирован, реализующий его модуль может быть выбран на этапе конфигурирования системы и будет встроен в систему статически, без каких-либо накладных расходов.
При этом важным свойством данной ОС является то, что она является «открытой» системой, то есть допускает использование сторонних модулей, без изменения существующих. Для добавления нового алгоритма планирования достаточно просто скопировать его реализацию в папку исходных текстов ОС. Такие модули могут разрабатываться независимыми разработчиками, всё что им нужно знать – это имена интерфейсов которые они импортируют и четкая спецификация экспортируемых сервисов. Отсутствие связей между компонентами позволяет распространять такие компоненты как независимые модули и использовать их в разных проектах без изменений.

Заключение

Операционные системы плохо поддаются разбиению на модули. В классической микроядерной архитектуре это разбиение происходит во время выполнения, что неприемлемо для встраиваемых систем по соображениям производительности. Подход к построению встраиваемых ОС в корне отличается от подхода к построению ОС общего назначения, так как к последним не предъявляется требований по широкому охвату классов целевых систем и «адаптации» специально для конкретного приложения. Внедрение зависимости может быть эффективно использовано для построения конфигурируемых ОС для встраиваемых систем, так как оно обеспечивает конфигурирование на уровне исходных текстов, что обеспечивает отсутствие накладных расходов и особенно важно для ограниченных по памяти и вычислительным ресурсам встроенных систем.