Модуль 1 . Ядро Linux

Введение

История Linux

Хотя Linux, по всей видимости, является самой популярной операционной системой с открытым исходным кодом, на самом деле ее история в сравнении с другими операционными системами относительно коротка. На заре компьютерной эры программисты разрабатывали свои программы для "голой" аппаратуры, используя языки, понятные для этой аппаратуры. В отсутствие операционной системы использовать всю большую и дорогую вычислительную машину в каждый конкретный момент времени могло только одно приложение (и один пользователь). Первые операционные системы были разработаны в 1950-е годы, чтобы облегчить жизнь разработчиков. В качестве примера можно назвать General Motors Operating System (GMOS), разработанную для IBM 701, и FORTRAN Monitor System (FMS), созданную North American Aviation для IBM 709.

В 1960-е годы в Массачусетском Технологическом институте (MIT) и в ряде компаний была разработана экспериментальная операционная система Multics (Multiplexed Information and Computing Service) для машины GE-645. Один из разработчиков этой ОС, компания AT&T, отошла от Multics и в 1970 году разработала свою собственную систему Unics. Вместе с этой ОС поставлялся язык C. При этом C был разработан и написан так, чтобы обеспечить переносимость разработки операционной системы.

Двадцать лет спустя Эндрю Танненбаум (Andrew Tanenbaum) создал микроядерную версию UNIX® под названием MINIX (minimal UNIX): совместимая с UNIX операционная система для персональных компьютеров, которая загружалась с дискет и умещалась в очень ограниченной в те времена памяти персонального компьютера. MINIX был создан Эндрю Таненбаумом в качестве учебной операционной системы, демонстрирующей архитектуру и возможности UNIX, но непригодной для полноценной работы с точки зрения программиста. Эта операционная система с открытым исходным кодом вдохновила Линуса Торвальдса (Linus Torvalds) на разработку первой версии Linux в начале 1990-х. Название своему ядру он дал freax, но позже оно было изменено хозяином ftp сервера на Linux — гибрид имени создателя и слова UNIX.

Совместимость с UNIX на тот момент означала, что операционная система должна поддерживать стандарт POSIX. POSIX — это функциональная модель совместимой с UNIX операционной системы, в которой описано, как должна вести себя система в той или иной

ситуации, но не приводится никаких указаний, как это следует реализовать программными средствами. POSIX описывал те свойства UNIX-совместимых систем, которые были общими для разных реализаций UNIX на момент создания этого стандарта. В частности, в POSIX описаны системные вызовы, которые должна обрабатывать операционная система, совместимая с этим стандартом.

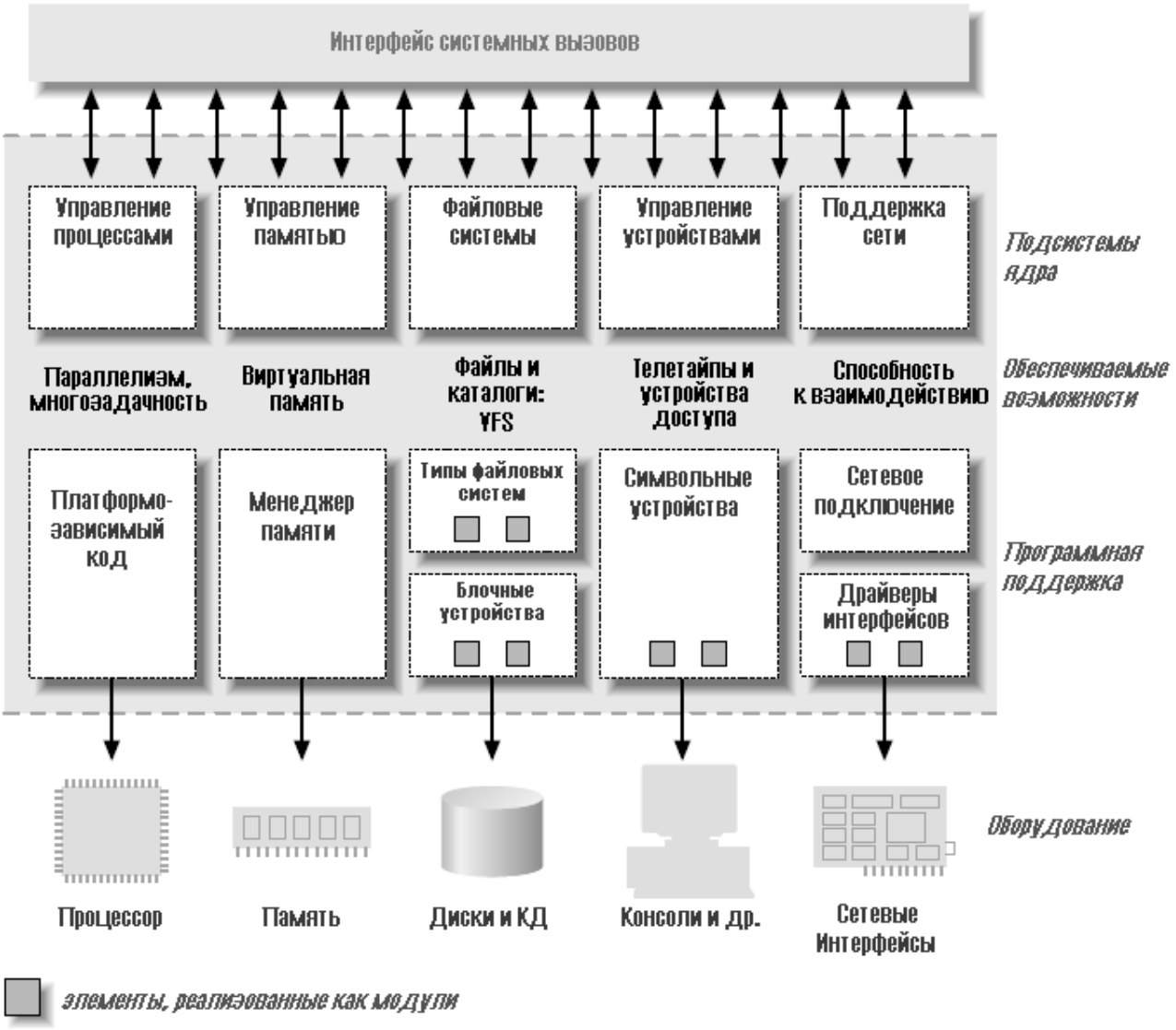
Linux быстро превратился из инициативы энтузиаста-одиночки во всемирный проект, в котором участвуют тысячи разработчиков. Одним из важнейших решений в судьбе Linux стало принятие лицензии GNU General Public License (GPL). GPL защитила ядро Linux от коммерческой эксплуатации и одновременно открыла путь к использованию разработок сообщества пользователей проекта GNU, основанного Ричардом Столлменом (Richard Stallman), объемы кода которого значительно превосходят даже объем ядра Linux. Это позволило использовать в Linux такие полезные приложения, как комплекс компиляторов GNU Compiler Collection (GCC) и различные командные оболочки.

Драйвера в ОС Linux

Драйверам устройств отводится особая роль в ядре Linux. Это “чёрные ящики”, которые обеспечивают доступ к различным устройствам посредством унифицированного программного интерфейса; они полностью скрывают детали того, как работает устройство. Этот программный интерфейс таков, что драйверы могут быть собраны отдельно от остальной части ядра и подключены в процессе работы, когда это необходимо. Такая модульность делает драйверы Linux простыми для написания, так что теперь доступны сотни драйверов.

Архитектура ядра linux

ОС Linux - многопользовательская многозадачная ОС. Несколько параллельных процессов обслуживают разные задачи. Каждый процесс запрашивает системные ресурсы, будь то процессорное время, память, сетевое подключение, или какие-то другие ресурсы. Ядро - это большая сложная программа, отвечающего за обработку таких запросов. Хотя границы между разными задачами ядра не всегда ясно определены, роль ядра может быть разделена на следующие части:



Управление процессами

Ядро отвечает за создание и уничтожение процессов и обеспечение их взаимодействия с внешним миром (ввод и вывод). Взаимодействие между разными процессами (через сигналы, каналы или примитивы межпроцессных взаимодействий) является основой общей функциональности системы и также возложена на ядро. Дополнительно, планировщик, который распределяет время

процессора, тоже является частью системы управления процессами. В общих словах, деятельность процессов управления ядра создаёт абстракцию нескольких процессов поверх одного или нескольких процессоров.

Управление памятью

Память компьютера - главный ресурс и способ управления ей особенно важен для производительности системы. Ядро создаёт виртуальное адресное пространство для каждого процесса поверх имеющихся ограниченных ресурсов. Разные части ядра взаимодействуют с подсистемой управления памятью через набор функциональных вызовов, начиная от простой пары malloc/free до много более развитой функциональности.

Файловые системы

Unix очень сильно связана с концепцией файловой системы; почти всё в Unix может быть обработано как файл. Ядро строит структурированную файловую систему поверх неструктурированного оборудования и полученная файловая абстракция интенсивно используется всей системой. В дополнение Linux поддерживает множество типов файловых систем, то есть различные способы организации данных на физическом носителе. К примеру, диски могут быть отформатированы в стандартной для Linux файловой системе ext3, часто используемой файловой системе FAT или некоторых других.

Управление устройствами

Почти каждая системная операция в конечном счёте связывается с физическим устройством. За исключением процессора, памяти и очень немногих других объектов, каждая операция управления устройством выполняются кодом, специфичным для данного адресуемого устройства. Этот код называется драйвером устройства. Ядро должно иметь встроенный драйвер устройства для каждой периферии, существующей в системе, от жесткого диска до клавиатуры и ленточного накопителя. Этот аспект функциональности ядра и является нашим основным интересом в этой книги.

Сетевые подключения

Сетевые подключения должно управляться операционной системой, потому что большинство сетевых операций не зависят от процессов: входящие пакеты - это асинхронные события. Эти пакеты должны быть собраны, распознаны и распределены перед тем, как они будут переданы другому процессу для обработки. Система отвечает за доставку пакетов данных между программой и сетевыми интерфейсами и должно управлять исполнением программ в

зависимости от их сетевой активности. Дополнительно, все задачи маршрутизации и разрешение адресов встроены в ядро.

Реализацию ядра Linux можно разделить на три больших уровня. Наверху располагается интерфейс системных вызовов, который реализует базовые функции, например, чтение и запись. Ниже интерфейса системных вызовов располагается код ядра, точнее говоря, архитектурно-независимый код ядра. Этот код является общим для всех процессорных архитектур, поддерживаемых Linux. Еще ниже располагается архитектурно-зависимый код, образующий т.н. BSP (Board Support Package - пакет поддержки аппаратной платформы). Этот код зависит от процессора и платформы для конкретной архитектуры.

Отличия разработки драйверов от прикладного ПО

Во-первых, самое важное отличие для программиста, пишущего код модуля: ядро не имеет доступа к стандартным библиотекам языка C (как, собственно, и к любым другим библиотекам). Для этого есть несколько причин - скорость выполнения, объем кода и т.п. А как следствие, ядро оперирует со своим собственным набором API, отличающимся от POSIX API (отличающимся по набору функций, по их наименованиям, по их прототипам, параметрам, и т.д.). Это видно на примере идентичных по смыслу, но различающихся вызовов printk() и printf(). Более подробно мы рассмотрим это на следующем занятии. И если и будут иногда встречаться якобы идентичные функции (strlen(), strcat() и многие другие), то это только внешняя видимость совпадения. Эти функции реализуют ту же функциональность, но это дубликатная реализация: подобный код реализуется и размещается в разных местах: для POSIX API в составе библиотек, а для модулей — в составе ядра.

Второе отличие - это отсутствие защиты памяти. Если обычная программа предпринимает попытку некорректного обращения к памяти, то ядро аварийно завершит процесс, послав ему сигнал SIGSEGV. Если же само ядро предпримет попытку некорректного обращения к памяти, последствия могут быть более тяжелыми. К тому же ядро не использует замещение страниц: каждый байт, используемый в ядре – это один байт физической памяти.

В-третьих, в ядре нельзя использовать вычисления с плавающей точкой. Активизация режима вычислений с плавающей точкой требует (при переключении контекста) сохранения и восстановления регистров устройства поддержки вычислений с плавающей точкой (FPU). Вряд ли в модуле ядра могут понадобиться вещественные вычисления, но если такое и случится, то их нужно эмулировать через целочисленные вычисления (для этого существует множество библиотек, из которых может быть заимствован исходный код).

В-четвертых, код ядра, включая код модулей, должен быть реентера́бельным (от англ. reentrant — повторно входимый), т.е. одна и та же копия инструкций программы в памяти может быть совместно использована несколькими пользователями или процессами. В общем случае, для обеспечения реентерабельности необходимо, чтобы вызывающий процесс или функция каждый раз передавал вызываемому процессу все необходимые данные. Таким образом, функция, которая: 1) зависит только от своих параметров, 2) не использует глобальные и статические переменные и 3) вызывает только реентерабельные функции, будет реентерабельной. Если

функция использует глобальные или статические переменные, необходимо обеспечить, чтобы каждый пользователь хранил свою локальную копию этих переменных.

Последнее отличие – это фиксированный стек (область адресного пространства, в которой выделяются локальные переменные). Локальные переменные – это все переменные, объявленные внутри любого программного блока, начинающегося сразу за левой открывающей фигурной скобкой, и не имеющие ключевого слова static. Стек в режиме ядра ограничен по размеру и не может быть изменён. Поэтому в коде ядра нужно крайне осмотрительно использовать (или не использовать вообще) конструкции, сокращающие пространство стека: рекурсивные вызовы, передавать структуру в функцию в качестве параметра «по значению» или возвращать структуру из функции, объявление крупных локальных структур внутри функций и т.д. Обычно стек равен двум страницам памяти, что например для x86, соответствует 8 КБ для 32 разрядных систем и 16 КБ для 64 разрядных.

Потенциальные проблемы с безопасностью

Если в ядре есть «уязвимость» в безопасности, тогда это ставит под угрозу безопасность системы в целом. Много текущих проблем безопасности происходят из-за ошибок, связанных с переполнением буфера, которые часто возникают из-за того, что программист забывает сравнить объем данных, записываемых в буфер, с его размером и переполняет его данными, что может позволить злоумышленнику внедрить свой код в адресное пространство ядра системы.

Любой полученный ввод информации от процесса пользователя должен быть обработан с огромным подозрением, никогда не надейтесь на него пока не проверите все. Будьте внимательны с неинициализированной памятью, любая память перед использованием должна быть обнулена или другим образом проинициализирована до того, как будет доступна для пользовательского процесса или устройства. Иначе результатом может оказаться утечка информации (обнаружение информации, паролей и т.д.).

Лицензирование модулей ядра

Linux лицензируется по версии 2 GNU General Public License (GPL), документа, разработанного для проекта GNU Фондом бесплатного программного обеспечения. GPL позволяет любому распространять и даже продавать продукт, покрытый GPL, пока получатель имеет доступ к исходнику и в состоянии реализовать те же самые права. Дополнительно, любой программный продукт, произошедший от продукта, покрытого GPL, если он вообще распространяется, должен быть выпущен под GPL.

- GPL — GNU Public License v2+
- GPL v2 — GNU Public License v2 only
- GPL and additional rights
- Dual BSD/GPL — BSD or GPL на выбор
- Dual MPL/GPL — MPL or GPL на выбор
- Proprietary — source code is not open