Роберт Лав - Linux.Системное программирование.2008.djvu

1. Введение.

Общие слова

В последние годы наблюдается такая тенденция, что прикладное программирование отходит от программирования на системном уровне и превращается в высокоуровневую разработку с использованием либо веб-приложений (таких, как JavaScript и PHP), либо управляемого кода (например, на C# или Java).

Такая разработка, однако, не предвещает смерть системного программирования. Действительно, кому-то же нужно писать интерпретатор JavaScript и среду исполнения C#, а это тоже программирование. Помимо этого, разработчики, пишущие на PHP или Java, также могут извлекать выгоду из знания системного программирования, так как понимание базовых элементов позволяет писать лучший код, неважно, на каком уровне он создается.

Несмотря на эту тенденцию в прикладном программировании, большая часть кода для Unix и Linux все так же пишется на системном уровне. Обычно это код на языке C, существующий в основном на базе интерфейсов, предоставляемых библиотекой C и ядром.

Вершиной системного программирования зачастую называют разработку ядра или, по крайней мере, написание драйверов устройств. Мы сфокусируемся на программировании системного уровня в пользовательском пространстве, то есть на всем, что лежит выше ядра (хотя знание внутреннего устройства ядра может стать полезным дополнением). Написание драйверов устройств и сетевое программирование — это огромные и постоянно расширяющиеся темы, которые рассматриваются в узкоспециализированных книгах.

Существует три краеугольных камня, на которых держится системное программирование в Linux: системные вызовы, библиотека C и компилятор C. Каждый из них заслуживает официального представления.

Системные вызовы

Системное программирование начинается с системных вызовов. Системные вызовы — это обращения к функциям, которые делаются из пользовательского пространства — текстового редактора, вашей любимой игры и т. д. — в ядро (главный внутренний орган системы) для того, чтобы запросить у операционной системы какую-либо службу или ресурс.

В Linux реализовано намного меньше системных вызовов, чем в ядрах большинства других операционных систем. Следовательно, системные вызовы, имеющиеся в одной архитектуре, могут отличаться от списка системных вызовов в другой архитектуре. Тем не менее очень большой поднабор системных вызовов — более 90 % — реализован одинаково во всех архитектурах.

Выполнение системных вызовов

Невозможно напрямую связать приложения пользовательского пространства с пространством ядра. По причинам, относящимся к безопасности и надежности, приложения пользовательского пространства не могут непосредственно выполнять код ядра или манипулировать данными ядра. Вместо этого предоставляется механизм, при помощи которого приложение пользовательского пространства может «сигнализировать» ядру, что ему необходимо выполнить системный вызов. Приложение может отправить

прерывание ядру и исполнить только тот код, к которому ему разрешает обращаться ядро. Конкретная реализация механизма варьируется в зависимости от архитектуры.

Общий принцип такой. Приложение выполняет инструкцию программного прерывания (int 0x80), после чего происходит переход в пространство ядра — защищенную область ядра, где ядро исполняет обработчик программного прерывания (обработчик системного вызова).

Приложение сообщает ядру, какой системный вызов нужно выполнить и с какими параметрами, используя аппаратные регистры (machine register).

Системные вызовы обозначаются номерами, начиная с нуля.

Например, вызов exit() – это 1. Тогда 1 помещается, например, в регистр eax, а после это выполняется int. Передача параметров обрабатывается схожим образом. Каждый параметр кладется в свой регистр(ebx, ecx, edx, esi и edi). В редких случаях, когда системный вызов включает в себя более пяти параметров. Если параметров больше 5, тогда один из регистров указывает на буфер в пользовательском пространстве, где хранятся все параметры вызова.

Пример,

mov eax, 1

mov ebx, 0

int 0x80.

mov eax, 1 - в EAX помещается 1 - номер системного вызова “exit”

mov ebx, 0 - в EBX помещается 0 - параметр вызова "exit", который означает код с которым завершится выполнение программы int 0x80 - системный вызов.

После системного вызова "exit" выполнение программы завершается.

Как системному программисту, обычно вам не нужно знать, каким способом ядро обрабатывает выполнение системных вызовов. Это знание закодировано в стандартных соглашениях о вызовах для каждой конкретной архитектуры и автоматически обрабатывается компилятором и библиотекой C.

Библиотека C

Библиотека C (libc) находится в самом сердце приложений Unix. Даже когда вы программируете на другом языке, библиотека C, вероятнее всего, все равно принимает участие в работе, обернутая более высокоуровневыми библиотеками, и предоставляет корневые службы, упрощая выполнение системных вызовов. В современных системах Linux библиотека C — это GNU libc, сокращенно glibc.

Помимо реализации стандартной библиотеки C, glibc предоставляет обертки для системных вызовов, поддержку многопоточной обработки и базовые возможности приложений.

Компилятор C

В Linux стандартный компилятор C предоставляется в форме коллекции компиляторов GNU (GNU Compiler Collection, gcc). Изначально названный GNU C Compiler поддерживал только язык Си. Позднее GCC был расширен для компиляции исходных кодов на таких языках программирования, как C++, Objective-C, Java, Фортран и Ada.

Поэтому сегодня gcc используется в качестве общего имени для семейства компиляторов GNU. Однако gcc — это также двоичный файл, при помощи которого

вызывается компилятор C. Далее, говоря о gcc, обычно будет иметься в виду программа gcc, если контекст не подразумевает иное.

Компилятор, используемый в системах Unix, в том числе в Linux, тесно связан с системным программированием, так как компилятор помогает реализовывать стандарт C и системный интерфейс ABI.

Интерфейсы API и ABI

Программисты, естественно, заинтересованы в том, чтобы их программы выполнялись во всех системах, поддержка которых декларируется, и не только сегодня, но и в будущем. Им необходима уверенность в том, что программы, которые они пишут для своих дистрибутивов Linux, будут работать и в других дистрибутивах, а также в прочих поддерживаемых архитектурах Linux и в более новых (или старых) версиях Linux.

На системном уровне существует два отдельных набора определений и описаний, влияющих на переносимость программ. Первый из них — это интерфейс прикладного программирования (Application Programming Interface, API), а второй — двоичный интерфейс приложений (Application Binary Interface, ABI). Оба определяют и описывают интерфейсы между различными частями компьютерного программного обеспечения.

Интерфейсы API

В интерфейсе API определяются способы, при помощи которых один фрагмент программного обеспечения общается с другим на уровне исходных текстов. Он предоставляет абстракцию в виде стандартного набора интерфейсов — обычно функций, которые какая-то часть программного обеспечения (обычно, хотя и не всегда, высокоуровневая) может вызывать из другой части программного обеспечения (обычно низкоуровневой). Например, API может абстрагировать концепцию вывода текста на экран при помощи семейства функций, обеспечивающих все необходимое для отображения текста. В API просто определяется интерфейс. Фрагмент программного обеспечения, который фактически предоставляет интерфейс API, называется реализацией API (implementation). Очень часто API называют «контрактом». Это не совсем верно, но крайней мере в юридическом смысле термина, так как API — это не двухстороннее соглашение. Пользователь API (обычно высокоуровневое программное обеспечение) не может ничего внести в API и его реализацию. Он может использовать API в том виде, в котором интерфейс существует, или же вообще не использовать его — третьего не дано! API всего лишь гарантирует, что если обе части программного обеспечения будут удовлетворять требованиям API, то они будут совместимы на уровне исходного кода (source compatible). Это означает, что приложение-пользователь API будет успешно компилироваться с данной реализацией API.

Интерфейсы ABI

Тогда как API определяет исходный интерфейс, ABI определяет низкоуровневый двоичный интерфейс между двумя или несколькими частями программного обеспечения в конкретной архитектуре. Интерфейс ABI формулирует, как приложение взаимодействует с самим собой, как приложение взаимодействует с ядром и как приложение взаимодействует с библиотекой. ABI обеспечивает совместимость на двоичном уровне (binary compatibility), то есть гарантирует, что фрагмент конечной программы будет функционировать в любой системе, где имеется тот же ABI, не требуя перекомпиляции. Интерфейсы ABI связаны с такими вещами, как соглашения о вызовах, порядок следования байтов, использование регистров, выполнение системных вызовов, компоновка, поведение библиотек и формат двоичных объектов. Системный программист должен быть знаком с ABI, но обычно необходимости запоминать подробности не

возникает. ABI реализуется цепочкой инструментов (toolchain) — компилятором, компоновщиком и т. д. — и обычно не выходит на поверхность никакими другими способами. Знание ABI, однако, может помочь научиться создавать оптимальные программы и необходимо для написания кода сборки или для того, чтобы влезть в саму цепочку инструментов (что также относится к системному программированию).

Описание ABI для любой конкретной архитектуры в Linux можно найти в Интернете; этот интерфейс реализуется цепочкой инструментов соответствующей архитектуры и ядром.

Стандарты

Системное программирование в Unix — это древнее искусство. Основы программирования в Unix остаются неприкосновенными вот уже десятилетия. Системы Unix, однако, весьма динамичны, и их поведение меняется с добавлением новых функций. Для того чтобы внести свой вклад в процесс превращения порядка в хаос, группы стандартов объединяются в официальные стандарты, включающие в себя варианты системных интерфейсов. Существует множество подобных стандартов, но, технически говоря, Linux официально не удовлетворяет ни одному из них. Однако система Linux нацелена на соответствие двум наиболее важным и распространенным стандартам: POSIX и Single UNIX Specification (SUS — единая спецификация Unix).

Стандарты POSIX и SUS документируют, среди прочего, API языка C для интерфейса Unix-подобной операционной системы. Фактически, они определяют системное программирование или, но крайней мере, самую общеупотребительную его часть для совместимых систем Unix.

Далее будем рассматривать функции, основанные на этих двух стандартах.

Файлы и файловая система

Файл — это самая простая и фундаментальная абстракция в Linux. Linux придерживается философии «все есть файл». Следовательно, большая часть взаимодействия реализуется через считывание и запись файлов, даже когда интересующий вас объект никак нельзя назвать файлом в общепринятом смысле этого слова.

Для того чтобы обратиться к файлу, его сначала нужно открыть. Файл можно открыть для чтения, записи или для чтения и записи одновременно. Обращение к открытому файлу осуществляется с использованием уникального дескриптора, обеспечивающего отображение метаданных, связанных с открытым файлом, на сам файл. Внутри ядра Linux этот дескриптор обрабатывается при помощи целочисленного значения (типа int в C), называемого дескриптором файла (file descriptor) или сокращенно fd. Файловые дескрипторы совместно используются пользовательским пространством и ядром, а пользовательские программы применяют их, чтобы напрямую обращаться к файлам. Большая доля системного программирования в Linux состоит из открытия, манипулирования, закрытия и использования дескрипторов файлов различными способами.

Обычные файлы

То, что большинство из нас называют «файлами», в Linux носит название обычного файла (regular file). Обычный файл содержит байты данных, организованные в линейный массив, который называется потоком байтов.

Основные понятия: позиция в файле (file position) или смещение в файле (file offset), длина файла и имя файла. Каждому открытому файлу приписывается свой дескриптор. Один и тот же файл может быть открыт несколько раз, например в разных потоках.

Хотя доступ к файлам обычно осуществляется при помощи имен файлов (filename), в действительности файлы напрямую не связываются со своими именами. Вместо этого обращение к файлу производится через структуру inode (первоначально information node, информационный узел), которой присваивается уникальное числовое значение. Это значение называется номером inode (inode number); и чаще всего его название сокращают до i-number или ino. В inode хранятся связанные с файлом метаданные, такие, как его временная метка, указывающая момент последней модификации, его владелец, тип, длина и местоположение данных файла — но не имя файла! Inode — это и физический объект, который можно обнаружить в Unix-подобных файловых системах, и умозрительная сущность, представляемая структурой данных в ядре Linux.

Каталоги и ссылки

Обращаться к файлу при помощи его номера inode затруднительно (и это потенциальная брешь в безопасности), поэтому из пользовательского пространства файлы всегда открываются по имени, а не по номеру inode. Каталоги (directory) используются для определения имен, по которым выполняется обращение к файлам. Каталог играет роль отображения между удобными для человека именами и номерами inode. Пара из имени и номера inode называется ссылкой (link). Физическая форма этого отображения на диске — простая таблица, хэш или что-либо еще — реализуется и управляется кодом ядра, который поддерживает данную файловую систему. По существу, каталог — это обычный файл, но его отличие от обычного файла состоит в том, что он содержит только карту отображения имен в номера inode. Ядро напрямую использует эти отображения для разрешения имен в номера inode.

Когда приложение из пользовательского пространства запрашивает открытие определенного имени файла, ядро открывает каталог, в котором хранится данное имя файла, и ищет указанное имя. Исходя из имени файла, ядро получает номер inode. Inode содержит метаданные, связанные с файлом, включая местоположение данных файла на диске.

Первоначально на диске есть только один каталог, корневой каталог (root directory). Этот каталог обычно обозначается путем /. Но, как мы все знаем, чаще всего в системе хранится много каталогов. Так как же ядро узнает, в какой каталог нужно заглянуть, чтобы найти определенное имя файла?

Как я уже сказано выше, каталоги очень похожи на обычные файлы. И с ними также связаны структуры inode. Следовательно, ссылки внутри каталогов могут указывать на inode других каталогов. Это означает, что каталоги могут вкладываться друг в друга, формируя иерархию каталогов, что, в свою очередь, позволяет использовать полные пути (pathname), с которыми знакомы все пользователи Unix, например: /home/blackbeard/landscaping.txt.

Когда ядро получает запрос на открытие такого полного пути, оно проходит по всем записям каталога (directory entry; внутри ядра — dentry) в пути, каждый раз находя inode следующей записи. В предыдущем примере ядро начинает просмотр с каталога /, получает inode для каталога home, переходит туда, получает inode для blackbeard, переходит в этот каталог и, наконец, получает inode для landscaping.txt. Эта операция называется разрешением каталога, или разрешением полного пути (directory, или pathname

resolution). Ядро Linux также использует кэш, называемый кэшем dentry (dentry cache), для хранения результатов разрешения каталогов, что обеспечивает более быстрый поиск в будущем, с учетом сосредоточенности во времени.

Полный путь, начинающийся с корневого каталога, является полностью уточненным (fully qualified) и называется абсолютным полным путем.

Некоторые полные пути не полностью уточнены; они указываются по отношению к какому-то другому каталогу (например, todo/plunder). Такие пути называются относительными полными путями. Если ядру предоставляется относительный полный путь, то разрешение его начинается с текущего рабочего каталога (current working directory). В текущем рабочем каталоге ядро ищет каталог todo. Попав туда, ядро получает inode для каталога plunder.

Хотя каталоги обрабатываются как обычные файлы, ядро не позволяет открывать их и манипулировать ими как обычными файлами. Работать с каталогами можно, только используя специальный набор системных вызовов. Такие системные вызовы позволяют добавлять и удалять ссылки — все равно только эти две операции и имеют смысл. Если бы пользовательскому пространству было разрешено манипулировать каталогами без посредничества ядра, то файловая система могла бы быть полностью разрушена из-за единственной ошибки.

Жесткие ссылки

Жесткие ссылки позволяют создавать в файловой системе сложные структуры, в которых несколько полных имен указывают на одни и те же данные. Жесткие ссылки могут находиться в одном каталоге или в нескольких разных каталогах. В любом случае ядро просто разрешает полный путь в правильную inode. Например, две жесткие ссылки — /home/bluebeard/map.txt и /home/blackbeard/treasure.txt — могут разрешать определенный номер inode в один и тот же фрагмент данных на диске. При удалении файла выполняется отсоединение его от структуры каталогов, и делается это всего лишь путем удаления из каталога пары из имени и inode, соответствующей этому файлу. Так как Linux поддерживает жесткие ссылки, файловая система не может разрушать inode и связанные данные при каждой операции отсоединения. Что, если где-то в файловой системе существует другая жесткая ссылка? Для того чтобы гарантировать, что файл не будет разрушен до тех пор, пока все ссылки на него не будут удалены, каждая структура inode содержит счетчик ссылок (link count), позволяющий отслеживать в системе число ссылок, указывающих на нее. Когда полное имя отсоединяется, счетчик ссылок уменьшается на единицу; только когда он достигает нуля, inode и связанные данные фактически удаляются из файловой системы.

Символические ссылки

Символические ссылки могут указывать на разные файловые системы. Эти ссылки оказывают большую нагрузку на систему, чем жесткие ссылки, потому что разрешение символической ссылки фактически включает в себя разрешение двух файлов: самой символической ссылки, а затем файла, на который она указывает. Так называемые ярлыки.

Специальные файлы (special file)

файлы блочных устройств. Доступ осуществляется через массив байтов (Пример, жесткие диски, дисководы гибких дисков, приводы компакт-дисков и flash-память);

файлы устройств посимвольного ввода-вывода. Доступ к устройству посимвольного ввода-вывода осуществляется как к линейной очереди байтов. Пример, клавиатура.

именованные конвейеры (IPC). это механизм взаимодействия процессов, обеспечивающий коммуникационный канал через файловый дескриптор, доступ к которому выполняется через специальный файл.

доменные сокеты Unix. Сокеты представляют собой расширенную форму IPC, обеспечивающую коммуникацию междудвумя различными процессами, причем не только на одной машине, но и на двух разных машинах.

Процессы

Если файлы — это самая фундаментальная абстракция в системе Unix, то сразу за ними следуют процессы. Процессы (process) — это выполняющийся код конечных программ: активные, живые, работающие программы. Но процессы — это больше чем просто конечные программы. Процессы состоят из данных, ресурсов, состояния и виртуализированного процессора.

Потоки выполнения

Каждый процесс состоит из одного или нескольких потоков выполнения (thread of execution), часто называемых просто потоками (thread). Поток — это единица активности в процессе, абстракция, несущая ответственность за исполнение кода и поддержание состояния выполнения процесса.

Иерархия процессов

Каждый процесс идентифицируется уникальным положительным целым числом, называемым идентификатором процесса (process ID, pid). Pid первого процесса равен единице, и каждый последующий процесс получает новое уникальное значение pid. В Linux процессы формируют жесткую иерархию, известную как дерево процессов.

У каждого процесса, за исключением первого, есть предок. Если предок — родительский процесс — завершается до того, как заканчивается выполнение его потомка, то ядро переназначает предка (reparent) для потомка, делая его потомком процесса инициализации.

Когда процесс завершается, он не удаляется немедленно из системы. Наоборот, ядро сохраняет в памяти часть резидента процесса, чтобы предок процесса мог запросить состояние своего потомка после завершения. Это называется обслуживанием (waiting on) завершенного процесса. Как только родительский процесс обслужит завершенный дочерний процесс, дочерний процесс полностью удаляется. Процесс, который уже завершился, но еще не был обслужен, называется зомби (zombie). Как положено, процесс инициализации обслуживает все свои дочерние процессы, гарантируя, что процессы с переназначенным предком не превратятся навсегда в зомби.

Пользователи и группы. Разрешения Не будем заострять внимание.

Команда ls.

Таблица 1.1. Биты разрешений и их значения

Бит	Восьмеричное значение	Текстовое значение	Соответствующее разрешение
8	400	r———	Владелец может читать
7	200	-w——-	Владелец может писать
6	100	–x——	Владелец может выполнять
5	040	––r—-	Группа может читать
4	020	—w—	Группа может писать
3	010	—-x–-	Группа может выполнять
2	004	——r–	Кто угодно может читать
1	002	——-w-	Кто угодно может писать
0	001	———x	Кто угодно может выполнять

Сигналы

Сигналы — это механизм односторонних асинхронных уведомлений. Сигнал может отправляться ядром процессу, процессом другому процессу или процессом самому себе. Сигналы обычно извещают процесс о каком-то событии, таком, как сбой сегментации, или таком, как нажатие пользователем клавишного сочетания Ctrl+C.

Обработка ошибок

В системном программировании ошибка обозначается при помощи возвращаемого значения функции и описывается специальной переменной errno. Дальше в основном будем использовать этот механизм. Эта переменная определена в заголовке <errno.h>. Пример, if(func()==-1) printf(“%d”, errno);

Определение препроцессора	Описание
E2BIG	Слишком длинный список аргументов
EACCESS	Отсутствует разрешение
EAGAIN	Повторите попытку
EBADF	Неправильный номер файла
EBUSY	Устройство или ресурс заняты
ECHILD	Дочерние процессы отсутствуют
EDOM	Математический аргумент за пределами домена функции
EEXIT	Файл уже существует
EFAULT	Неправильный адрес
EFBIG	Слишком большой файл
EINTR	Системный вызов был прерван
EINVAL	Недопустимый аргумент
EIO	Ошибка ввода-вывода
EISDIR	Это каталог
EMFILE	Слишком много открытых файлов
EMLINK	Слишком много ссылок
ENFILE	Переполнение таблицы файлов

Определение препроцессора	Описание
ENODEV	Устройство отсутствует
ENOENT	Файл или каталог отсутствует
ENOEXEC	Ошибка формата запуска
ENOMEM	Нехватка памяти
ENOSPC	На устройстве нет свободного пространства
ENOTDIR	Это не каталог
ENOTTY	Недопустимая операция управления ввода-вывода
ENXIO	Устройство или адрес отсутствует
EPERM	Операция не разрешена
EPIPE	Сломанный конвейер
ERANGE	Слишком большой результат
EROFS	Файловая система, доступная только для чтения
ESPIPE	Недопустимый поиск
ESRCH	Процесс отсутствует
ETXTBSY	Текстовый файл занят
EXDEV	Неправильная ссылка

Как преобразовать значение errno в текстовый вид.

#include <stdio.h>

*void perror (const char *str);* строковое представление текущей ошибки, взятое из errno, добавляя в качестве префикса строку, на которую указывает параметр str, за которой следует двоеточие.

Пример, if (close (fd) == -1) perror("close").

#include <string.h>

*char * strerror (int errnum);*Тут все понятно.

int strerror_r (int errnum, char *buf, size_t len); Заполняет buf длинной len. Возращает 0 в случае завершения. -1 в случае сбоя.

Errno глобально по всему потоку, так как любой вызов любой функции может изменить значение, поэтому так не правильно:

```
if (fsync (fd) == -1) {

fprintf (stderr. "fsync failed!\n").

if (errno == EIO)

fprintf (stderr, "I/O error on %d", fd);

}
```

fsync копирует на диск все части файла, находящиеся в памяти и ожидает пока устройство скажет, что все эти части сохранены.

EBADF fd не является дескриптором файла, открытого для записи.

EROFS, **EINVAL**
fd связан со специальным файлом, который не поддерживает синхронизацию.

EIO Во время синхронизации произошла ошибка.

Нужно так:

```
if (fsync (fd) == -1) {

int err= errno;

fprintf (stderr. "fsync failed!\n").

if (err == EIO)

fprintf (stderr, "I/O error on %d", fd);

}
```

2. **Файловый ввод-вывод**
3. **Управления файлами и каталогами**
4. **Управления процессами**
5. **Управления памятью**
6. **Сигналы**