Эмуляция Linux® в FreeBSD

Аннотация

Эта магистерская диссертация посвящена обновлению слоя эмуляции Linux® (так называемого *Linuxulator*). Задача состояла в обновлении слоя для соответствия функциональности Linux® 2.6. В качестве эталонной реализации было выбрано ядро Linux® 2.6.16. Концепция основана на реализации NetBSD. Большая часть работы была выполнена летом 2006 года в рамках программы Google Summer of Code для студентов. Основное внимание уделялось добавлению поддержки *NPTL* (новой библиотеки потоков POSIX®) в слой эмуляции, включая *TLS* (локальное хранилище потоков), *фьютексы (futex)* (быстрые мьютексы в пользовательском пространстве), *PID mangling* и некоторые другие второстепенные аспекты. В процессе было выявлено и исправлено множество мелких проблем. Моя работа была интегрирована в основной репозиторий исходного кода FreeBSD и войдет в предстоящий релиз 7.0R. Мы, команда разработчиков эмуляции, работаем над тем, чтобы сделать эмуляцию Linux® 2.6 стандартным слоем эмуляции в FreeBSD.

Содержание

1. Введение 1
2. Взгляд изнутри… 2
3. Эмуляция 11
4. Слой эмуляции Linux® - машинозависимая часть 17
5. Слой эмуляции Linux® - машинонезависимая часть 21
6. Заключение 33
7. Литература 35

1. Введение

В последние несколько лет операционные системы с открытым исходным кодом на основе UNIX® начали широко использоваться на серверных и клиентских машинах. Среди этих операционных систем я хотел бы выделить две: FreeBSD — за наследие BSD, проверенную временем кодобазу и множество интересных возможностей, и Linux® — за широкую пользовательскую базу, активное сообщество разработчиков и поддержку крупных компаний. FreeBSD чаще используется на серверных машинах, выполняющих сложные сетевые задачи, и реже — на настольных компьютерах обычных пользователей. В то время как Linux® также применяется на серверах, он гораздо популярнее среди домашних пользователей. Это приводит к ситуации, когда для Linux® доступно множество проприетарных программ, которые не поддерживают FreeBSD.

Естественно, возникает необходимость в возможности запуска Linux® бинарников в системе FreeBSD, и именно этому посвящена данная работа: эмуляции ядра Linux® в

операционной системе FreeBSD.

Летом 2006 года компания Google Inc. спонсировала проект, направленный на расширение слоя эмуляции Linux® (так называемого Linuxulator) в FreeBSD для включения возможностей Linux® 2.6. Данная диссертация написана в рамках этого проекта.

2. Взгляд изнутри...

В этом разделе мы рассмотрим каждую из рассматриваемых операционных систем. Как они работают с системными вызовами, фреймами прерываний и другими низкоуровневыми аспектами. Также мы опишем, как они интерпретируют общие примитивы UNIX®, такие как PID, потоки и т. д. В третьем подразделе мы поговорим о том, как в целом может быть реализована эмуляция UNIX® на UNIX®.

2.1. Что такое UNIX®

UNIX® — это операционная система с долгой историей, которая повлияла почти на все остальные операционные системы, используемые в настоящее время. Начиная с 1960-х годов, её разработка продолжается и по сей день (хотя в разных проектах). Вскоре развитие UNIX® разделилось на два основных направления: семейства BSD и System III/V. Они взаимно влияли друг на друга, формируя общий стандарт UNIX®. Среди вклада, возникшего в BSD, можно назвать виртуальную память, сетевой стек TCP/IP, FFS и многие другие. Ветка System V внесла свой вклад в примитивы межпроцессного взаимодействия SysV, копирование при записи и т. д. Самого UNIX® больше не существует, но его идеи были использованы многими другими операционными системами по всему миру, образовав так называемые UNIX®-подобные операционные системы. В наши дни наиболее влиятельными из них являются Linux®, Solaris и, возможно (в некоторой степени), FreeBSD. Существуют корпоративные производные UNIX® (AIX, HP-UX и т. д.), но они всё больше мигрируют на упомянутые системы. Давайте подведём итог типичным характеристикам UNIX®.

2.2. Технические детали

Каждая запущенная программа представляет собой процесс, который отражает состояние вычислений. Выполняющийся процесс разделяется между пространством ядра и пользовательским пространством. Некоторые операции могут выполняться только из пространства ядра (например, работа с оборудованием), но процесс должен проводить большую часть своего времени в пользовательском пространстве. Ядро — это место, где происходит управление процессами, оборудованием и низкоуровневыми деталями. Ядро предоставляет стандартный унифицированный UNIX® API для пользовательского пространства. Наиболее важные из них рассмотрены ниже.

2.2.1. Обмен данными между ядром и пользовательским процессом

Общий API UNIX® определяет системный вызов как способ передачи команд из пользовательского процесса ядру. Наиболее распространённая реализация использует либо прерывание, либо специализированную инструкцию (например, инструкции `SYSENTER` /`SYSCALL` для ia32). Системные вызовы определяются по номеру. Например, в FreeBSD

системный вызов номер 85 — это swapon(2), а номер 132 — mkfifo(2). Некоторые системные вызовы требуют параметров, которые передаются из пользовательского пространства в пространство ядра различными способами (зависит от реализации). Системные вызовы являются синхронными.

Еще один возможный способ взаимодействия — использование *прерывания*. Прерывания происходят асинхронно после возникновения определённого события (деление на ноль, ошибка страницы и т.д.). Прерывание может быть прозрачным для процесса (ошибка страницы) или привести к реакции, например, отправке *сигнала* (деление на ноль).

2.2.2. Обмен данными между процессами

Существуют другие API (System V IPC, разделяемая память и т.д.), но наиболее важным API являются сигналы. Сигналы отправляются процессами или ядром и принимаются процессами. Некоторые сигналы могут быть проигнорированы или обработаны пользовательской процедурой, другие приводят к предопределённому действию, которое нельзя изменить или игнорировать.

2.2.3. Управление процессами

Процессы ядра обрабатываются первыми в системе (так называемый init). Каждый запущенный процесс может создать свою идентичную копию, используя системный вызов fork(2). Были введены некоторые немного изменённые версии этого системного вызова, но базовая семантика остаётся той же. Каждый запущенный процесс может превратиться в другой процесс, используя системный вызов exec(3). Были введены некоторые модификации этого системного вызова, но все они служат одной и той же базовой цели. Процессы завершают своё существование, вызывая системный вызов exit(2). Каждый процесс идентифицируется уникальным номером, называемым PID. У каждого процесса есть определённый родитель (идентифицируемый его PID).

2.2.4. Управление потоками

Традиционный UNIX® не определяет никакого API или реализации для потоков, в то время как POSIX® определяет свой API для потоков, но реализация остаётся неопределённой. Традиционно существовало два способа реализации потоков: обработка их как отдельных процессов (потоки 1:1) или обёртывание всей группы потоков в один процесс с управлением потоками в пользовательском пространстве (потоки 1:N). Сравнение основных особенностей каждого подхода:

Потоки 1:1

- тяжеловесные потоки
- планирование не может быть изменено пользователем (частично смягчено благодаря POSIX® API) + нет необходимости в обёртке системных вызовов + может использовать несколько процессоров

Потоки 1:N

+ легковесные потоки + планирование может быть легко изменено пользователем -

Системные вызовы должны быть обернуты - не может использовать более одного CPU

2.3. Что такое FreeBSD?

Проект FreeBSD — одна из старейших операционных систем с открытым исходным кодом, доступных для повседневного использования. Она является прямым потомком оригинальной UNIX®, поэтому можно утверждать, что это настоящая UNIX®, хотя проблемы с лицензированием не позволяют этого сделать. Начало проекта относится к началу 1990-х годов, когда группа пользователей BSD создала набор исправлений для операционной системы 386BSD. На основе этого набора возникла новая операционная система под названием FreeBSD, получившая своё имя благодаря либеральной лицензии. Другая группа создала операционную систему NetBSD с другими целями. Мы сосредоточимся на FreeBSD.

FreeBSD — это современная операционная система на основе UNIX®, обладающая всеми возможностями UNIX®. Вытесняющая многозадачность, многопользовательские функции, сетевые возможности TCP/IP, защита памяти, поддержка симметричной многопроцессорности, виртуальная память с объединёнными VM и кэшем буфера — всё это присутствует. Одной из интересных и чрезвычайно полезных особенностей является возможность эмуляции других UNIX®-подобных операционных систем. По состоянию на декабрь 2006 года и разработку 7-CURRENT поддерживаются следующие функции эмуляции:

- Совместимость FreeBSD/i386 на FreeBSD/amd64
- FreeBSD/i386 эмуляция на FreeBSD/ia64
- Эмуляция Linux® операционной системы Linux® на FreeBSD
- NDIS-эмуляция интерфейса сетевых драйверов Windows
- NetBSD-эмуляция операционной системы NetBSD
- Поддержка PECoff для исполняемых файлов FreeBSD в формате PECoff
- Эмуляция SVR4 System V revision 4 UNIX®

Активно разрабатываемые эмуляции — это слой Linux® и различные слои FreeBSD-on-FreeBSD. Остальные в настоящее время не должны работать корректно или быть пригодными к использованию.

2.3.1. Технические детали

FreeBSD — это традиционный вариант UNIX® в смысле разделения выполнения процессов на две части: выполнение в пространстве ядра и выполнение в пространстве пользователя. Существует два типа входа процесса в ядро: системный вызов (syscall) и ловушка (trap). Возврат только один. В последующих разделах мы опишем три входа/выхода в/из ядра. Всё описание относится к архитектуре i386, так как Linuxulator существует только там, но концепция схожа на других архитектурах. Информация была взята из [1] и исходного кода.

Системные записи

В FreeBSD существует абстракция, называемая загрузчиком классов исполнения, которая

является прослойкой в системном вызове execve(2). Она использует структуру `sysentvec`, описывающую ABI исполняемого файла. Эта структура содержит такие элементы, как таблицу преобразования errno, таблицу преобразования сигналов, различные функции для обработки системных вызовов (исправление стека, создание дампов памяти и т.д.). Каждый ABI, который ядро FreeBSD поддерживает, должен определять эту структуру, так как она используется в дальнейшем в коде обработки системных вызовов и в некоторых других местах. Системные вызовы обрабатываются обработчиками прерываний, где можно одновременно получить доступ как к пространству ядра, так и к пользовательскому пространству.

Системные вызовы

Системные вызовы в FreeBSD выполняются путем прерывания `0x80` с установленным в регистре `%eax` номером нужного системного вызова и аргументами, переданными через стек.

Когда процесс вызывает прерывание `0x80`, срабатывает обработчик системного вызова `int0x80` (определённый в sys/i386/i386/exception.s), который подготавливает аргументы (т.е. копирует их в стек) для вызова функции на языке C syscall(2) (определённой в sys/i386/i386/trap.c), обрабатывающей переданный фрейм прерывания. Обработка включает подготовку системного вызова (в зависимости от записи `sysvec`), определение разрядности системного вызова (32-битный или 64-битный, что влияет на размер параметров), после чего параметры копируются, включая сам системный вызов. Затем выполняется фактическая функция системного вызова с обработкой кода возврата (особые случаи для ошибок `ERESTART` и `EJUSTRETURN`). В завершение планируется вызов `userret()`, возвращающий процесс в пользовательское пространство. Параметры для фактического обработчика системного вызова передаются в виде аргументов `struct thread *td`, `struct syscall args *`, где второй параметр является указателем на скопированную структуру параметров.

Ловушки (trap)

Обработка ловушек в FreeBSD аналогична обработке системных вызовов. При возникновении ловушки вызывается обработчик на ассемблере. Он выбирается между `alltraps`, `alltraps` с сохранением регистров или `calltrap` в зависимости от типа ловушки. Этот обработчик подготавливает аргументы для вызова функции на языке C `trap()` (определена в sys/i386/i386/trap.c), которая затем обрабатывает произошедшую ловушку. После обработки она может отправить сигнал процессу и/или вернуться в пользовательское пространство с помощью `userret()`.

Выходы

Выход из ядра в пользовательское пространство происходит с использованием ассемблерной процедуры `doreti`, независимо от того, было ли ядро вызвано через ловушку или через системный вызов. Это восстанавливает состояние программы из стека и возвращает управление в пользовательское пространство.

Примитивы UNIX®

Операционная система FreeBSD придерживается традиционной схемы UNIX®, где каждый

процесс имеет уникальный идентификационный номер, так называемый *PID* (Идентификатор Процесса). Номера PID выделяются либо линейно, либо случайным образом в диапазоне от `0` до `PID_MAX`. Распределение номеров PID осуществляется с помощью линейного поиска в пространстве PID. Каждый поток в процессе получает тот же номер PID в результате вызова getpid(2).

В настоящее время в FreeBSD существует два способа реализации потоков. Первый способ — это M:N потоки, за которым следует модель потоков 1:1. По умолчанию используется библиотека M:N (`libpthread`), но во время выполнения можно переключиться на потоки 1:1 (`libthr`). Планируется в ближайшее время перейти на библиотеку 1:1 по умолчанию. Хотя обе библиотеки используют одни и те же примитивы ядра, доступ к ним осуществляется через разные API. Библиотека M:N использует семейство системных вызовов `kse_*`, тогда как библиотека 1:1 использует семейство `thr_*`. Из-за этого отсутствует общая концепция идентификатора потока, разделяемая между ядром и пользовательским пространством. Конечно, обе библиотеки реализуют API идентификатора потока pthread. У каждого потока ядра (как описано в `struct thread`) есть идентификатор td tid, но он недоступен напрямую из пользовательского пространства и служит исключительно нуждам ядра. Он также используется в библиотеке потоков 1:1 в качестве идентификатора потока pthread, но обработка этого идентификатора внутренняя для библиотеки и не может быть использована напрямую.

Как упоминалось ранее, в FreeBSD существуют две реализации потоков. Библиотека M:N разделяет работу между пространством ядра и пользовательским пространством. Поток — это сущность, которая планируется в ядре, но может представлять различное количество пользовательских потоков. M пользовательских потоков отображаются на N потоков ядра, что позволяет экономить ресурсы, сохраняя при этом возможность использовать преимущества многопроцессорного параллелизма. Дополнительную информацию о реализации можно получить из man-страницы или [1]. Библиотека 1:1 напрямую отображает пользовательский поток на поток ядра, что значительно упрощает схему. Ни одна из этих реализаций не включает механизм справедливости (такой механизм был реализован, но недавно удалён, поскольку вызывал серьёзное замедление и усложнял работу с кодом).

2.4. Что такое Linux®

Linux® — это UNIX®-подобное ядро, изначально разработанное Линусом Торвальдсом, а сейчас развиваемое множеством программистов по всему миру. От своих скромных начал до сегодняшнего дня, при широкой поддержке таких компаний, как IBM или Google, Linux® ассоциируется с быстрым темпом разработки, полной поддержкой оборудования и моделью организации по принципу "доброжелательного диктатора".

Разработка Linux® началась в 1991 году как любительский проект в Университете Хельсинки, Финляндия. С тех пор она приобрела все черты современной ОС, подобной UNIX®: многопроцессорность, поддержка многопользовательского режима, виртуальная память, сетевое взаимодействие — в общем, всё необходимое. Также присутствуют высокоуровневые функции, такие как виртуализация и т. д.

В 2006 году Linux®, похоже, был наиболее широко используемой открытой операционной

системой с поддержкой независимых поставщиков программного обеспечения, таких как Oracle, RealNetworks, Adobe и других. Большая часть коммерческого программного обеспечения, распространяемого для Linux®, доступна только в бинарном виде, поэтому перекомпиляция для других операционных систем невозможна.

Большая часть разработки Linux® происходит в системе контроля версий Git. Git — это распределённая система, поэтому нет централизованного источника кода Linux®, но некоторые ветви считаются основными и официальными. Схема нумерации версий, используемая в Linux®, состоит из четырёх чисел: A.B.C.D. В настоящее время разработка ведётся в ветке 2.6.C.D, где C обозначает мажорную версию, в которую добавляются или изменяются функции, а D — минорную версию, предназначенную только для исправления ошибок.

Дополнительную информацию можно получить из [3].

2.4.1. Технические детали

Linux® следует традиционной схеме UNIX®, разделяя выполнение процесса на две части: ядро и пользовательское пространство. Ядро может быть вызвано двумя способами: через ловушку (trap) или через системный вызов. Возврат осуществляется только одним способом. Далее описание относится к Linux® 2.6 на архитектуре i386™. Эта информация взята из [2].

Системные вызовы

Системные вызовы в Linux® выполняются (в пользовательском пространстве) с использованием макросов `syscallX`, где X заменяется числом, представляющим количество параметров данного системного вызова. Этот макрос преобразуется в код, который загружает регистр `%eax` номером системного вызова и выполняет прерывание `0x80`. После этого вызывается возврат из системного вызова, который преобразует отрицательные значения возврата в положительные значения `errno` и устанавливает `res` в `-1` в случае ошибки. При вызове прерывания `0x80` процесс переходит в ядро в обработчик ловушки системного вызова. Эта процедура сохраняет все регистры в стеке и вызывает выбранную точку входа системного вызова. Обратите внимание, что соглашение о вызовах Linux® предполагает передачу параметров системного вызова через регистры, как показано здесь:

1. параметр → `%ebx`
2. параметр → `%ecx`
3. параметр → `%edx`
4. параметр → `%esi`
5. параметр → `%edi`
6. параметр → `%ebp`

Существуют некоторые исключения из этого правила, где Linux® использует другие соглашения о вызовах (наиболее примечателен системный вызов `clone`).

Ловушки (trap)

Обработчики ловушек представлены в файле arch/i386/kernel/traps.c, а большинство этих обработчиков находятся в arch/i386/kernel/entry.S, где происходит обработка ловушек.

Выходы

Возврат из системного вызова обрабатывается функцией `syscall exit(3)`, которая проверяет, есть ли у процесса незавершённые задачи, затем проверяет, использовались ли селекторы, предоставленные пользователем. Если это произошло, применяется исправление стека, и, наконец, регистры восстанавливаются из стека, а процесс возвращается в пользовательское пространство.

Примитивы UNIX®

В версии 2.6 операционная система Linux® переопределила некоторые традиционные примитивы UNIX®, в частности PID, TID и поток. PID определяется не как уникальный для каждого процесса, поэтому для некоторых процессов (потоков) getppid(2) возвращает одинаковое значение. Уникальная идентификация процесса обеспечивается TID. Это связано с тем, что *NPTL* (New POSIX® Thread Library) определяет потоки как обычные процессы (так называемая модель 1:1). Создание нового процесса в Linux® 2.6 происходит с использованием системного вызова `clone` (варианты fork перереализованы с его использованием). Этот системный вызов clone определяет набор флагов, которые влияют на поведение процесса клонирования в отношении реализации потоков. Семантика немного размыта, так как нет единого флага, указывающего системному вызову создать поток.

Реализованные флаги клонирования:

- `CLONE_VM` - процессы разделяют общее адресное пространство
- `CLONE_FS` — совместно использовать umask, текущий рабочий каталог и пространство имён
- `CLONE_FILES` - совместно использовать открытые файлы
- `CLONE_SIGHAND` - разделять обработчики сигналов и заблокированные сигналы
- `CLONE_PARENT` - использовать один процесс к качестве родительского
- `CLONE_THREAD` — быть потоком (дальнейшие пояснения ниже)
- `CLONE_NEWNS` - новое пространство имен
- `CLONE_SYSVSEM` - совместное использование структур отмены SysV
- `CLONE_SETTLS` - настройка TLS по указанному адресу
- `CLONE_PARENT_SETTID` - установить TID в родителе
- `CLONE_CHILD_CLEARTID` - очистить TID в дочернем процессе
- `CLONE_CHILD_SETTID` - установить TID в дочернем процессе

`CLONE_PARENT` устанавливает реального родителя в родителя вызывающего процесса. Это полезно для потоков, потому что если поток A создаёт поток B, мы хотим, чтобы поток B был привязан к родителю всей группы потоков. `CLONE_THREAD` делает то же самое, что `CLONE_PARENT`,

CLONE_VM и CLONE_SIGHAND, перезаписывает PID, чтобы он совпадал с PID вызывающего процесса, устанавливает сигнал завершения в "нет" и входит в группу потоков. CLONE_SETTLS настраивает записи GDT для обработки TLS. Набор флагов CLONE_*_*TID устанавливает/сбрасывает предоставленный пользователем адрес в TID или 0.

Как видно, CLONE_THREAD выполняет большую часть работы и не очень хорошо вписывается в схему. Первоначальный замысел неясен (даже для авторов, согласно комментариям в коде), но я думаю, изначально был один флаг для потоков, который затем был разделён на множество других флагов, но это разделение так и не было завершено. Также непонятно, для чего нужно это разделение, так как glibc не использует его, и только ручное использование clone позволяет программисту получить доступ к этим возможностям.

Для непоточных программ PID и TID совпадают. Для поточных программ первый поток имеет одинаковые PID и TID, а каждый созданный поток разделяет тот же PID и получает уникальный TID (поскольку передаётся CLONE_THREAD), также родительский процесс общий для всех процессов, образующих эту поточную программу.

Код, реализующий pthread_create(3) в NPTL, определяет флаги clone следующим образом:

```
int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGNAL

 | CLONE_SETTLS | CLONE_PARENT_SETTID

| CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
#if __ASSUME_NO_CLONE_DETACHED == 0

| CLONE_DETACHED
#endif

| 0);
```

CLONE_SIGNAL определен как

```
#define CLONE_SIGNAL (CLONE_SIGHAND | CLONE_THREAD)
```

последний 0 означает, что сигнал не отправляется при завершении любого из потоков.

2.5. Что такое эмуляция

Согласно словарному определению, эмуляция — это способность программы или устройства имитировать другую программу или устройство. Это достигается за счёт предоставления той же реакции на заданный стимул, что и у эмулируемого объекта. На практике в мире программного обеспечения в основном встречаются три типа эмуляции — программа, используемая для эмуляции машины (QEMU, различные эмуляторы игровых консолей и т.д.), программная эмуляция аппаратного обеспечения (эмуляторы OpenGL, эмуляция блоков плавающей запятой и т.д.) и эмуляция операционной системы (либо в ядре операционной системы, либо в виде программы пользовательского пространства).

Эмуляция обычно используется в тех случаях, когда применение оригинального компонента невозможно или нецелесообразно. Например, может возникнуть необходимость использовать программу, разработанную для другой операционной системы. В такой ситуации на помощь приходит эмуляция. Иногда эмуляция — единственный возможный вариант, например, когда необходимое аппаратное устройство ещё не существует или уже не выпускается. Такое часто происходит при переносе операционной системы на новую (ещё не существующую) платформу. Иногда эмуляция просто экономически выгоднее.

С точки зрения реализации, существует два основных подхода к эмуляции. Вы можете либо эмулировать всё целиком — принимать возможные входные данные исходного объекта, поддерживать внутреннее состояние и выдавать корректные выходные данные на основе состояния и/или входных данных. Такой вид эмуляции не требует каких-либо специальных условий и, в принципе, может быть реализован где угодно для любого устройства/программы. Недостаток в том, что реализация такой эмуляции довольно сложна, трудоёмка и подвержена ошибкам. В некоторых случаях можно использовать более простой подход. Представьте, что вы хотите эмулировать принтер, печатающий слева направо, на принтере, который печатает справа налево. Очевидно, что нет необходимости в сложном слое эмуляции — достаточно просто перевернуть печатаемый текст. Иногда эмулирующая среда очень похожа на эмулируемую, и тогда достаточно тонкого слоя преобразования для обеспечения полностью рабочей эмуляции! Как видите, такой подход гораздо менее требователен к реализации, а значит, менее трудоёмок и подвержен ошибкам, чем предыдущий. Однако необходимое условие — две среды должны быть достаточно схожи. Третий подход сочетает в себе два предыдущих. Чаще всего объекты не предоставляют одинаковые возможности, поэтому в случае эмуляции более мощного объекта на менее мощном приходится эмулировать отсутствующие функции с помощью полной эмуляции, описанной выше.

Эта магистерская диссертация посвящена эмуляции UNIX® на UNIX®, что является именно тем случаем, когда достаточно тонкого слоя трансляции для обеспечения полной эмуляции. API UNIX® состоит из набора системных вызовов, которые обычно самодостаточны и не влияют на глобальное состояние ядра.

Существует несколько системных вызовов, которые влияют на внутреннее состояние, но это можно решить, предоставив некоторые структуры, поддерживающие дополнительное состояние.

Эмуляция не бывает идеальной, и в эмуляторах часто чего-то не хватает, но обычно это не вызывает серьёзных проблем. Представьте эмулятор игровой приставки, который эмулирует всё, кроме звука. Без сомнения, игры остаются играбельными, и эмулятором можно пользоваться. Возможно, это не так комфортно, как оригинальная приставка, но это приемлемый компромисс между ценой и удобством.

То же самое касается UNIX® API. Большинство программ могут работать с очень ограниченным набором системных вызовов. Эти вызовы, как правило, являются самыми старыми (read(2)/write(2), семейство fork(2), обработка signal(3), exit(3), API socket(2)), поэтому их легко эмулировать, поскольку их семантика одинакова во всех современных UNIX®-подобных системах.

3. Эмуляция

3.1. Как работает эмуляция в FreeBSD

Как упоминалось ранее, FreeBSD поддерживает выполнение бинарных файлов из нескольких других UNIX®-подобных систем. Это возможно благодаря наличию в FreeBSD абстракции, называемой загрузчик классов исполнения. Он интегрируется в системный вызов execve(2), поэтому когда execve(2) собирается выполнить бинарный файл, он анализирует его тип.

В FreeBSD существуют два основных типа исполняемых файлов. Текстовые скрипты, подобные shell-скриптам, которые идентифицируются по первым двум символам `#!`, и обычные (как правило, *ELF*) бинарные файлы, представляющие собой скомпилированные исполняемые объекты. Подавляющее большинство (можно сказать, все) исполняемых файлов в FreeBSD относятся к типу ELF. Файлы ELF содержат заголовок, который определяет ABI операционной системы для данного ELF-файла. Считывая эту информацию, операционная система может точно определить, к какому типу относится данный исполняемый файл.

Каждый ABI ОС должен быть зарегистрирован в ядре FreeBSD. Это относится и к родному ABI ОС FreeBSD. Таким образом, когда execve(2) выполняет двоичный файл, он перебирает список зарегистрированных API, и когда находит подходящий, начинает использовать информацию, содержащуюся в описании ABI ОС (его таблицу системных вызовов, таблицу преобразования `errno` и т.д.). Таким образом, каждый раз, когда процесс вызывает системный вызов, он использует свой собственный набор системных вызовов вместо какого-либо глобального. Это обеспечивает очень элегантный и простой способ поддержки выполнения различных двоичных форматов.

Природа эмуляции различных ОС (а также некоторых других подсистем) привела разработчиков к внедрению механизма обработчиков событий. В ядре существует множество мест, где вызывается список обработчиков событий. Каждая подсистема может зарегистрировать обработчик событий, и они вызываются соответствующим образом. Например, при завершении процесса вызывается обработчик, который может выполнить необходимую очистку для подсистемы.

Те простые средства предоставляют практически всё необходимое для инфраструктуры эмуляции, и, по сути, это единственное, что требуется для реализации слоя эмуляции Linux®.

3.2. Общие примитивы в ядре FreeBSD

Для работы слоев эмуляции требуется некоторая поддержка со стороны операционной системы. Я расскажу о некоторых поддерживаемых примитивах в операционной системе FreeBSD.

3.2.1. Примитивы синхронизации

Добавил: Attilio Rao <attilio@FreeBSD.org>

Примитивы синхронизации FreeBSD основаны на идее предоставления достаточно большого количества различных примитивов таким образом, чтобы для каждой конкретной подходящей ситуации можно было использовать наилучший.

На высоком уровне можно выделить три вида примитивов синхронизации в ядре FreeBSD:

- атомарные операции и барьеры памяти
- блокировки
- барьеры планирования

Ниже приведены описания для 3 семейств. Для каждой блокировки рекомендуется ознакомиться с соответствующей справочной страницей (где это возможно), чтобы получить более подробные объяснения.

Атомарные операции и барьеры памяти

Атомарные операции реализуются через набор функций, выполняющих простые арифметические действия над операндами в памяти атомарным образом по отношению к внешним событиям (прерываниям, вытеснению и т. д.). Атомарные операции могут гарантировать атомарность только для небольших типов данных (порядка величины типа `.long` в архитектуре C), поэтому их следует редко использовать напрямую в конечном коде, разве что для очень простых операций (например, установки флага в битовой карте). На самом деле довольно просто и часто можно допустить семантическую ошибку, полагаясь только на атомарные операции (обычно называемые lock-less). Ядро FreeBSD предоставляет способ выполнения атомарных операций в сочетании с барьерами памяти. Барьеры памяти гарантируют, что атомарная операция произойдет в определённом порядке относительно других обращений к памяти. Например, если нам нужно, чтобы атомарная операция выполнилась только после завершения всех ожидающих операций записи (с точки зрения переупорядочивания буферов инструкций), нам необходимо явно использовать барьер памяти вместе с этой атомарной операцией. Таким образом, легко понять, почему барьеры памяти играют ключевую роль в построении высокоуровневых блокировок (таких как refcounts, мьютексы и т. д.). Для подробного объяснения атомарных операций обратитесь к atomic(9). Однако важно отметить, что атомарные операции (и барьеры памяти тоже) в идеале должны использоваться только для построения фронтенд-блокировок (например, мьютексов).

Счетчики ссылок (refcount)

Счетчики ссылок (refcounts) — это интерфейсы для работы с подсчетом ссылок. Они реализованы с использованием атомарных операций и предназначены для случаев, когда счетчик ссылок — это единственное, что требует защиты, поэтому даже такие механизмы, как спин-мьютекс, не рекомендуются. Использование интерфейса refcount для структур, где уже применяется мьютекс, часто является ошибкой, так как, вероятно, следует защитить счетчик ссылок в рамках уже существующих защищенных участков кода. В настоящее время man-страница, посвященная refcount, отсутствует; для обзора существующего API

обратитесь к sys/refcount.h.

Блокировки

Ядро FreeBSD имеет множество классов блокировок. Каждая блокировка определяется некоторыми уникальными свойствами, но, вероятно, наиболее важным является событие, связанное с конкурирующими владельцами (или, другими словами, поведение потоков, неспособных захватить блокировку). Схема блокировок FreeBSD предлагает три различных поведения для конкурирующих потоков:

1. вращающиеся
2. блокирующие
3. спящие

номера приведены не случайно

Вращающиеся блокировки

Спин-блокировки позволяют ожидающим потокам продолжать работу (вращаться), пока они не смогут захватить блокировку. Важным аспектом является ситуация, когда поток соревнуется за спин-блокировку и не вытесняется. Поскольку ядро FreeBSD является вытесняющим, это подвергает спин-блокировки риску взаимоблокировок, которые можно устранить только отключением прерываний на время их удержания. По этой и другим причинам (таким как отсутствие поддержки распространения приоритетов, неэффективность схем балансировки нагрузки между CPU и т.д.), спин-блокировки предназначены для защиты очень небольших участков кода или, в идеале, не должны использоваться вовсе, если это не требуется явно (об этом далее).

Блокирующие

Блокирующие блокировки позволяют ожидающим потокам быть выгруженными и заблокированными до тех пор, пока владелец блокировки не освободит её и не разбудит один или несколько конкурентов. Чтобы избежать проблем с голоданием, блокирующие блокировки передают приоритет от ожидающих к владельцу. Блокирующие блокировки должны быть реализованы через интерфейс турникета и предназначены для наиболее частого использования в ядре, если нет особых условий.

Спящие

Спящие блокировки (с ожиданием) позволяют ожидающим потокам быть вытесненными и заснуть до тех пор, пока держатель блокировки не освободит её и не разбудит один или несколько ожидающих. Поскольку блокировки с ожиданием предназначены для защиты больших участков кода и обработки асинхронных событий, они не поддерживают распространение приоритетов. Они должны быть реализованы через интерфейс sleepqueue(9).

Порядок захвата блокировок очень важен, не только из-за возможности взаимоблокировки при обратном порядке захвата, но и потому, что захват блокировок должен следовать

определённым правилам, связанным с их природой. Если взглянуть на таблицу выше, практическое правило заключается в том, что если поток удерживает блокировку уровня n (где уровень — это число, указанное рядом с типом блокировки), ему запрещено захватывать блокировки более высоких уровней, так как это нарушит заданную семантику пути. Например, если поток удерживает блокирующую блокировку (уровень 2), ему разрешено захватывать спин-блокировку (уровень 1), но не спящую блокировку (уровень 3), поскольку блокирующие блокировки предназначены для защиты более коротких путей, чем спящие блокировки (однако эти правила не касаются атомарных операций или барьеров планирования).

Вот список блокировок с соответствующими типами поведения:

- spin mutex – вращающийся режим – mutex(9)
- sleep mutex – блокирующий режим – mutex(9)
- pool mutex – блокирующий режим – mtx(pool)
- Семейство функций sleep – спящий режим – sleep(9) pause tsleep msleep msleep_spin msleep_rw msleep_sx
- condvar – спящий режим – condvar(9)
- rwlock – блокирующий режим – rwlock(9)
- sxlock – спящий режим – sx(9)
- lockmgr – спящий режим – lockmgr(9)
- семафоры – спящий режим – sema(9)

Среди этих блокировок только мьютексы, sxlock, rwlock и lockmgr предназначены для обработки рекурсии, но в настоящее время рекурсия поддерживается только мьютексами и lockmgr.

Барьеры планирования

Барьеры планирования предназначены для управления планированием потоков. Они в основном состоят из трёх различных заглушек:

- критические секции (и вытеснение)
- sched_bind
- sched_pin

Как правило, их следует использовать только в определённом контексте, и даже если они часто могут заменять блокировки, их следует избегать, поскольку они не позволяют диагностировать простые потенциальные проблемы с помощью инструментов отладки блокировок (например, witness(4)).

Критические секции

В ядре FreeBSD была реализована вытесняющая многозадачность в основном для работы с потоками обработки прерываний. Фактически, чтобы избежать высокой задержки прерываний, потоки с приоритетом разделения времени могут быть вытеснены потоками

обработки прерываний (таким образом, им не нужно ждать планирования, как это предусмотрено в обычном случае). Однако вытеснение также вводит новые точки гонки, которые необходимо обрабатывать. Часто для борьбы с вытеснением проще всего полностью отключить его. Критическая секция определяет участок кода (ограниченный парой функций critical_enter(9) и critical_exit(9)), где гарантируется отсутствие вытеснения (пока защищённый код не будет полностью выполнен). Это часто может эффективно заменить блокировку, но должно использоваться осторожно, чтобы не потерять все преимущества, которые даёт вытеснение.

sched_pin/sched_unpin

Еще один способ работы с вытеснением — это интерфейс `sched_pin()`. Если участок кода заключен между функциями `sched_pin()` и `sched_unpin()`, гарантируется, что соответствующий поток, даже если он может быть вытеснен, всегда будет выполняться на том же CPU. Закрепление очень эффективно в частном случае, когда нам необходимо обращаться к данным, привязанным к определённым CPU, и мы предполагаем, что другие потоки не изменят эти данные. Последнее условие делает критическую секцию избыточно строгим условием для нашего кода.

sched_bind/sched_unbind

`sched_bind` — это API, используемый для привязки потока к определённому CPU на всё время выполнения кода, пока вызов функции `sched_unbind` не отменит эту привязку. Эта функция играет ключевую роль в ситуациях, когда нельзя доверять текущему состоянию CPU (например, на самых ранних этапах загрузки), так как требуется избежать миграции потока на неактивные CPU. Поскольку `sched_bind` и `sched_unbind` работают с внутренними структурами планировщика, их использование должно быть заключено в захват/освобождение `sched_lock`.

3.2.2. Структура proc

Различные уровни эмуляции иногда требуют дополнительных данных для каждого процесса. Можно управлять отдельными структурами (списком, деревом и т.д.), содержащими эти данные для каждого процесса, но это может быть медленно и потреблять много памяти. Чтобы решить эту проблему, структура `proc` в FreeBSD содержит `p_emuldata` — указатель типа void на данные, специфичные для уровня эмуляции. Эта запись `proc` защищена мьютексом proc.

Структура `proc` в FreeBSD содержит элемент `p_sysent`, который идентифицирует, под какой ABI работает данный процесс. Фактически, это указатель на упомянутый выше `sysentvec`. Таким образом, сравнивая этот указатель с адресом, по которому хранится структура `sysentvec` для данной ABI, мы можем эффективно определить, принадлежит ли процесс нашему эмуляционному слою. Код обычно выглядит следующим образом:

```
if (__predict_true(p->p_sysent != &elf_Linux(R)_sysvec))
      return;
```

Как видите, мы эффективно используем модификатор `__predict_true`, чтобы свести наиболее

распространённый случай (процесс FreeBSD) к простой операции возврата, сохраняя высокую производительность. Этот код следует преобразовать в макрос, поскольку в настоящее время он не очень гибкий, например, мы не поддерживаем эмуляцию Linux®64, а также процессы Linux® в формате A.OUT на архитектуре i386.

3.2.3. VFS

Подсистема VFS в FreeBSD очень сложна, но слой эмуляции Linux® использует лишь небольшую её часть через чётко определённый API. Она может работать как с vnode, так и с файловыми дескрипторами. Vnode представляет собой виртуальный vnode, то есть представление узла в VFS. Другое представление — это файловый дескриптор, который представляет открытый файл с точки зрения процесса. Файловый дескриптор может представлять сокет или обычный файл. Файловый дескриптор содержит указатель на свой vnode. Более одного файлового дескриптора могут указывать на один и тот же vnode.

namei

Функция namei(9) является центральной точкой входа для поиска и преобразования путей. Она проходит по пути шаг за шагом от начальной до конечной точки, используя функцию поиска, которая является внутренней для VFS. Системный вызов namei(9) может обрабатывать символьные ссылки, абсолютные и относительные пути. Когда путь ищется с помощью namei(9), он заносится в кэш имён. Это поведение можно отключить. Данная функция используется повсеместно в ядре, и её производительность крайне важна.

vn_fullpath

Функция vn_fullpath(9) предпринимает максимальные усилия для обхода кэша имён VFS и возвращает путь для заданного (заблокированного) vnode. Этот процесс ненадёжен, но в большинстве типичных случаев работает корректно. Ненадёжность обусловлена тем, что функция опирается на кэш VFS (она не обходит структуры на носителе), не работает с жёсткими ссылками и т.д. Данная процедура используется в нескольких местах Linuxulator.

Операции с vnode

- `fgetvp` - по заданным потоку и номеру файлового дескриптора возвращает связанный vnode
- vn_lock(9) - блокирует vnode
- `vn_unlock` - разблокирует vnode
- VOP_READDIR(9) - читает каталог, на который ссылается vnode
- VOP_GETATTR(9) - получает атрибуты файла или каталога, на который ссылается vnode
- VOP_LOOKUP(9) - выполняет поиск пути к заданному каталогу
- VOP_OPEN(9) - открывает файл, на который ссылается vnode
- VOP_CLOSE(9) - закрывает файл, на который ссылается vnode
- vput(9) - уменьшает счетчик использования для vnode и разблокирует его
- vrele(9) - уменьшает счетчик использования для vnode

- vref(9) - увеличивает счетчик использования для vnode

Операции обработчика файлов (handler)

- `fget` - для заданного потока и номера файлового дескриптора возвращает связанный обработчик файла и делает на него ссылку
- `fdrop` - освобождает ссылку на обработчик файлов
- `fhold` - ссылается на обработчик файла

4. Слой эмуляции Linux® - машинозависимая часть

В этом разделе рассматривается реализация слоя эмуляции Linux® в операционной системе FreeBSD. Сначала описывается машинозависимая часть, рассказывающая о том, как и где реализовано взаимодействие между пользовательским пространством и ядром. Рассматриваются системные вызовы, сигналы, ptrace, ловушки и исправление стека. Эта часть посвящена архитектуре i386, но написана в общем виде, поэтому другие архитектуры не должны сильно отличаться. Следующая часть — машинонезависимая часть Linuxulator. Этот раздел охватывает только i386 и обработку ELF. A.OUT устарел и не поддерживается.

4.1. Обработка системных вызовов

Обработка системных вызовов в основном реализована в файле linux_sysvec.c, который покрывает большинство процедур, указанных в структуре `sysentvec`. Когда процесс Linux®, выполняющийся на FreeBSD, делает системный вызов, общая процедура обработки системных вызовов вызывает linux prepsyscall для ABI Linux®.

4.1.1. Linux® prepsyscall

Linux® передаёт аргументы системных вызовов через регистры (поэтому на i386 ограничено 6 параметрами), тогда как FreeBSD использует стек. Подпрограмма Linux® `prepsyscall` должна копировать параметры из регистров в стек. Порядок регистров следующий: `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`. Однако это верно только для *большинства* системных вызовов. Некоторые (особенно `clone`) используют другой порядок, но это, к счастью, легко исправить, добавив фиктивный параметр в прототип `linux_clone`.

4.1.2. Как писать системные вызовы

Каждый системный вызов, реализованный в Linuxulator, должен иметь свой прототип с различными флагами в syscalls.master. Формат файла следующий:

```
...
    AUE_FORK STD        { int linux_fork(void); }
...
    AUE_CLOSE NOPROTO   { int close(int fd); }
```

```
...
```

Первый столбец представляет номер системного вызова. Второй столбец предназначен для поддержки аудита. Третий столбец обозначает тип системного вызова. Он может быть `STD`, `OBSOL`, `NOPROTO` или `UNIMPL`. `STD` — это стандартный системный вызов с полным прототипом и реализацией. `OBSOL` означает устаревший вызов и определяет только прототип. `NOPROTO` означает, что системный вызов реализован в другом месте, поэтому не требует добавления префикса ABI и т.д. `UNIMPL` означает, что системный вызов будет заменён на `nosys` (системный вызов, который просто выводит сообщение о том, что вызов не реализован, и возвращает `ENOSYS`).

Из файла syscalls.master скрипт генерирует три файла: linux_syscall.h, linux_proto.h и linux_sysent.c. Файл linux_syscall.h содержит определения имён системных вызовов и их числовых значений, например:

```
...
#define LINUX_SYS_linux_fork 2
...
#define LINUX_SYS_close 6
...
```

linux_proto.h содержит определения структур аргументов для каждого системного вызова, например:

```
struct linux_fork_args {
  register_t dummy;
};
```

И, наконец, linux_sysent.c содержит структуру, описывающую таблицу системных вызовов, используемую для фактической диспетчеризации системного вызова, например:

```
{ 0, (sy_call_t *)linux_fork, AUE_FORK, NULL, 0, 0 }, /* 2 = linux_fork */
{ AS(close_args), (sy_call_t *)close, AUE_CLOSE, NULL, 0, 0 }, /* 6 = close */
```

Как видно, `linux_fork` реализован в самом Linuxulator, поэтому определение имеет тип `STD` и не имеет аргументов, что демонстрируется структурой-заглушкой. С другой стороны, `close` — это просто псевдоним для настоящего FreeBSD close(2), поэтому у него нет связанной структуры аргументов Linux, и в системной таблице вызовов он не имеет префикса linux, так как вызывает настоящий close(2) в ядре.

4.1.3. Нереализованные системные вызовы

Слой эмуляции Linux® не является полным, так как некоторые системные вызовы реализованы неправильно, а некоторые не реализованы вовсе. В слое эмуляции используется механизм для пометки нереализованных системных вызовов с помощью

макроса DUMMY. Эти заглушки находятся в файле linux_dummy.c в форме DUMMY(syscall);, которые затем преобразуются в различные вспомогательные файлы системных вызовов, а их реализация сводится к выводу сообщения о том, что данный системный вызов не реализован. Прототип UNIMPL не используется, потому что мы хотим иметь возможность идентифицировать имя вызванного системного вызова, чтобы понимать, какие системные вызовы более важны для реализации.

4.2. Обработка сигналов

Обработка сигналов обычно выполняется в ядре FreeBSD для всех вариантов бинарной совместимости с помощью вызова уровня, зависящего от совместимости. Слой совместимости Linux® определяет для этой цели процедуру linux_sendsig.

4.2.1. Linux® sendsig

Эта процедура сначала проверяет, установлен ли сигнал с флагом SA_SIGINFO, в таком случае она вызывает процедуру linux_rt_sendsig вместо текущей. Далее она выделяет (или повторно использует уже существующий) контекст обработчика сигнала, затем формирует список аргументов для обработчика сигнала. Она преобразует номер сигнала на основе таблицы преобразования сигналов, назначает обработчик, преобразует sigset. Затем она сохраняет контекст для процедуры sigreturn (различные регистры, преобразованный номер trap и маску сигналов). Наконец, она копирует контекст сигнала в пользовательское пространство и подготавливает контекст для фактического выполнения обработчика сигнала.

4.2.2. linux_rt_sendsig

Эта процедура аналогична linux_sendsig, только подготовка контекста сигнала отличается. Она добавляет siginfo, ucontext и некоторые части POSIX®. Стоит рассмотреть возможность объединения этих двух функций с выгодой в виде меньшего дублирования кода и, возможно, даже более быстрого выполнения.

4.2.3. linux_sigreturn

Этот системный вызов используется для возврата из обработчика сигнала. Он выполняет некоторые проверки безопасности и восстанавливает исходный контекст процесса. Также он разблокирует сигнал в маске сигналов процесса.

4.3. Ptrace

Многие производные UNIX® реализуют системный вызов ptrace(2) для обеспечения различных функций отслеживания и отладки. Этот механизм позволяет трассирующему процессу получать различную информацию о трассируемом процессе, такую как дампы регистров, любую память из адресного пространства процесса и т.д., а также трассировать процесс, например, пошагово выполнять инструкции или между системными вызовами (сисколлами и ловушками). ptrace(2) также позволяет устанавливать различную информацию в трассируемом процессе (регистры и т.д.). ptrace(2) является стандартом для

UNIX®, реализованным в большинстве UNIX®-систем по всему миру.

Эмуляция Linux® в FreeBSD реализует механизм ptrace(2) в файле linux_ptrace.c. Функции для преобразования регистров между Linux® и FreeBSD и фактический системный вызов эмуляции ptrace(2). Системный вызов представляет собой длинный блок switch, который реализует свой аналог в FreeBSD для каждой команды ptrace(2). Команды ptrace(2) в основном одинаковы между Linux® и FreeBSD, поэтому обычно требуется лишь небольшая модификация. Например, `PT_GETREGS` в Linux® работает с непосредственными данными, в то время как FreeBSD использует указатель на данные, поэтому после выполнения (нативного) системного вызова ptrace(2) необходимо выполнить copyout для сохранения семантики Linux®.

Реализация ptrace(2) в Linuxulator имеет известные недостатки. Наблюдались паники при использовании `strace` (который является потребителем ptrace(2)) в среде Linuxulator. Также `PT_SYSCALL` не реализован.

4.4. Ловушки (trap)

Всякий раз, когда процесс Linux®, выполняющийся в слое эмуляции, вызывает прерывание (trap), само прерывание обрабатывается прозрачно, за исключением преобразования прерывания. Linux® и FreeBSD расходятся во мнениях относительно того, что является прерыванием, поэтому этот вопрос решается здесь. Код на самом деле очень короткий:

```
static int
translate_traps(int signal, int trap_code)
{

  if (signal != SIGBUS)
    return signal;

  switch (trap_code) {

    case T_PROTFLT:
    case T_TSSFLT:
    case T_DOUBLEFLT:
    case T_PAGEFLT:
      return SIGSEGV;

    default:
      return signal;
  }
}
```

4.5. Исправление стека

Динамический редактор связей RTLD ожидает так называемые AUX-теги на стеке во время выполнения `execve`, поэтому необходимо выполнить исправление, чтобы это обеспечить. Конечно, каждая система RTLD отличается, поэтому уровень эмуляции должен

предоставлять собственную процедуру исправления стека. Linuxulator делает именно это. Функция `elf_linux_fixup` просто копирует AUX-теги на стек и корректирует стек пользовательского процесса, чтобы он указывал сразу после этих тегов. Таким образом, RTLD работает умным способом.

4.6. Поддержка A.OUT

Эмуляционный слой Linux® на i386 также поддерживает бинарные файлы Linux® в формате A.OUT. Почти всё, что описано в предыдущих разделах, должно быть реализовано для поддержки A.OUT (кроме перевода ловушек и отправки сигналов). Поддержка бинарных файлов A.OUT больше не поддерживается, в частности, эмуляция 2.6 с ними не работает, но это не вызывает никаких проблем, так как linux-base в портах, вероятно, вообще не поддерживает бинарные файлы A.OUT. Эта поддержка, скорее всего, будет удалена в будущем. Большая часть кода, необходимого для загрузки бинарных файлов Linux® A.OUT, находится в файле imgact_linux.c.

5. Слой эмуляции Linux® - машинонезависимая часть

В этом разделе рассматривается машинонезависимая часть Linuxulator. Он охватывает инфраструктуру эмуляции, необходимую для эмуляции Linux® 2.6, реализацию thread local storage (TLS) (на i386) и фьютексы. Затем мы кратко обсуждаем некоторые системные вызовы.

5.1. Описание NPTL

Одним из основных направлений прогресса в разработке Linux® 2.6 стала поддержка потоков. До версии 2.6 поддержка потоков в Linux® реализовывалась в библиотеке linuxthreads. Эта библиотека представляла собой частичную реализацию потоков POSIX®. Потоки создавались как отдельные процессы с использованием системного вызова `clone`, что позволяло им разделять адресное пространство (и другие ресурсы). Основными недостатками такого подхода были разные PID для каждого потока, некорректная обработка сигналов (с точки зрения pthreads) и т.д. Кроме того, производительность оставляла желать лучшего (использование сигналов `SIGUSR` для синхронизации потоков, потребление ресурсов ядра и т.п.), поэтому для решения этих проблем была разработана новая система потоков под названием NPTL.

Библиотека NPTL была сосредоточена на двух вещах, но появилась третья, поэтому её обычно считают частью NPTL. Этими двумя вещами были встраивание потоков в структуру процесса и фьютекс. Дополнительной третьей вещью стал TLS, который не требуется напрямую NPTL, но вся пользовательская библиотека NPTL зависит от него. Эти улучшения привели к значительному росту производительности и соответствию стандартам. В настоящее время NPTL является стандартной библиотекой потоков в системах Linux®.

Реализация Linuxulator в FreeBSD подходит к NPTL в трёх основных направлениях: TLS, фьютекс и изменение PID, что предназначено для эмуляции потоков Linux®. В следующих

разделах описывается каждое из этих направлений.

5.2. Инфраструктура эмуляции Linux® 2.6

Эти разделы посвящены тому, как управляются потоки Linux® и как мы моделируем это в FreeBSD.

5.2.1. Определение эмуляции 2.6 во время выполнения

Слой эмуляции Linux® в FreeBSD поддерживает динамическую настройку эмулируемой версии. Это выполняется с помощью sysctl(8), а именно `compat.linux.osrelease`. Установка этого sysctl(8) влияет на поведение слоя эмуляции во время выполнения. При установке значения 2.6.x устанавливается переменная `linux_use_linux26`, а при установке другого значения она остаётся сброшенной. Эта переменная (а также аналогичные переменные для каждой клетки) определяет, используется ли в коде инфраструктура 2.6 (в основном, преобразование PID). Настройка версии применяется глобально для всей системы и влияет на все процессы Linux®. Не следует изменять sysctl(8) во время выполнения любого бинарного файла Linux®, так как это может привести к проблемам.

5.2.2. Идентификаторы процессов и потоков Linux®

Семантика потоков в Linux® немного запутанная и использует совершенно другую терминологию по сравнению с FreeBSD. Процесс в Linux® состоит из `struct task`, включающей два поля идентификаторов — PID и TGID. PID — это *не* идентификатор процесса, а идентификатор потока. TGID идентифицирует группу потоков, другими словами, процесс. Для однопоточного процесса PID равен TGID.

Поток в NPTL — это обычный процесс, у которого TGID не равен PID и есть групповой лидер, отличный от него самого (и, конечно, общая виртуальная память и т.д.). Все остальное происходит так же, как и с обычным процессом. Нет разделения общего состояния на внешнюю структуру, как в FreeBSD. Это создаёт некоторое дублирование информации и возможную несогласованность данных. Ядро Linux®, похоже, использует информацию о задаче → группе в одних местах и информацию о задаче в других, что не очень последовательно и выглядит небезопасно с точки зрения возможных ошибок.

Каждый поток NPTL создаётся вызовом системного вызова `clone` с определённым набором флагов (подробнее в следующем подразделе). NPTL реализует строгую модель потоков 1:1.

В FreeBSD мы эмулируем потоки NPTL с помощью обычных процессов FreeBSD, которые разделяют виртуальную память и т.д., а гимнастика с PID просто имитируется в специфической для эмуляции структуре, прикреплённой к процессу. Структура, прикреплённая к процессу, выглядит следующим образом:

```
struct linux_emuldata {
  pid_t pid;

  int *child_set_tid; /* in clone(): Child.s TID to set on clone */
  int *child_clear_tid;/* in clone(): Child.s TID to clear on exit */
```

```
  struct linux_emuldata_shared *shared;

  int pdeath_signal; /* parent death signal */

  LIST_ENTRY(linux_emuldata) threads; /* list of linux threads */
};
```

PID используется для идентификации процесса FreeBSD, к которому присоединена эта структура. `child_se_tid` и `child_clear_tid` используются для копирования адреса TID при завершении и создании процесса. Указатель `shared` указывает на структуру, разделяемую между потоками. Переменная `pdeath_signal` определяет сигнал завершения родительского процесса, а указатель `threads` используется для связывания этой структуры со списком потоков. Структура `linux_emuldata_shared` выглядит следующим образом:

```
struct linux_emuldata_shared {

  int refs;

  pid_t group_pid;

  LIST_HEAD(, linux_emuldata) threads; /* head of list of linux threads */
};
```

`refs` — это счётчик ссылок, используемый для определения момента, когда можно освободить структуру, чтобы избежать утечек памяти. `group_pid` служит для идентификации PID (= TGID) всего процесса (= группы потоков). Указатель `threads` является головой списка потоков в процессе.

Структуру `linux_emuldata` можно получить из процесса с помощью `em_find`. Прототип функции выглядит следующим образом:

```
struct linux_emuldata *em_find(struct proc *, int locked);
```

Здесь `proc` — это процесс, из которого мы хотим получить структуру `emuldata`, а параметр `locked` определяет, нужно ли блокировать. Допустимые значения — `EMUL_DOLOCK` и `EMUL_DOUNLOCK`. Подробнее о блокировке позже.

5.2.3. Преобразование PID

Поскольку между FreeBSD и Linux® существуют различия в представлении идентификатора процесса (PID) и идентификатора потока (TID), нам необходимо преобразовывать эти понятия. Это достигается за счёт модификации PID. Это означает, что мы изменяем представление о PID (=TGID) и TID (=PID) между ядром и пользовательским пространством. Основное правило заключается в следующем: в ядре (в Linuxulator) `PID = PID`, а `TGID = shared → group_pid`; для пользовательского пространства мы представляем `PID = shared → group_pid` и `TID = proc → p_pid`. Член `PID` в структуре `linux_emuldata` является FreeBSD PID.

Вышесказанное в основном влияет на системные вызовы getpid, getppid, gettid. В случаях, где мы используем PID/TGID соответственно. При копировании TID в `child_clear_tid` и `child_set_tid` мы копируем FreeBSD PID.

5.2.4. Системный вызов clone

`clone` — это системный вызов, с помощью которого создаются потоки в Linux®. Прототип системного вызова выглядит следующим образом:

```
int linux_clone(l_int flags, void *stack, void *parent_tidptr, int dummy,
void * child_tidptr);
```

Параметр `flags` указывает системному вызову, как именно процессы должны быть клонированы. Как описано выше, Linux® может создавать процессы, разделяющие различные ресурсы независимо, например, два процесса могут разделять файловые дескрипторы, но не виртуальную память и т.д. Последний байт параметра `flags` является сигналом завершения для вновь созданного процесса. Параметр `stack`, если он не `NULL`, указывает, где находится стек потока, а если он `NULL`, предполагается копирование при записи стека вызывающего процесса (т.е. делать то же, что делает обычная функция fork(2)). Параметр `parent_tidptr` используется как адрес для копирования PID процесса (т.е. идентификатора потока) после того, как процесс достаточно инициализирован, но ещё не готов к выполнению. Параметр `dummy` присутствует из-за очень странного соглашения о вызовах этого системного вызова на i386. Он использует регистры напрямую и не позволяет компилятору делать это, что приводит к необходимости использования фиктивного системного вызова. Параметр `child_tidptr` используется как адрес для копирования PID после завершения ветвления процесса и при его завершении.

Системный вызов продолжает выполнение, устанавливая соответствующие флаги в зависимости от переданных аргументов. Например, `CLONE_VM` преобразуется в RFMEM (общее адресное пространство) и т.д. Единственная тонкость здесь — это `CLONE_FS` и `CLONE_FILES`, поскольку FreeBSD не позволяет устанавливать их отдельно, поэтому мы эмулируем это, не устанавливая RFFDG (копирование таблицы файловых дескрипторов и другой информации о файловой системе), если задан любой из этих флагов. Это не вызывает проблем, так как эти флаги всегда устанавливаются вместе. После установки флагов процесс создаётся с помощью внутренней процедуры `fork1`, при этом процесс настраивается так, чтобы не помещаться в очередь выполнения (т.е. не становиться исполняемым). После завершения ветвления мы, при необходимости, изменяем родителя для нового процесса, чтобы эмулировать семантику `CLONE_PARENT`. Следующий шаг — создание данных эмуляции. Потоки в Linux® не отправляют сигналы своим родителям, поэтому мы устанавливаем сигнал завершения в 0, чтобы отключить эту возможность. Затем выполняется настройка `child_set_tid` и `child_clear_tid`, что активирует соответствующую функциональность далее в коде. На этом этапе мы копируем PID по адресу, указанному в `parent_tidptr`. Установка стека процесса выполняется простой перезаписью регистра `%esp` (`%rsp` на amd64) в кадре потока. Далее настраивается TLS для нового процесса. После этого может быть эмулирована семантика vfork(2), и, наконец, новый процесс помещается в очередь выполнения, а его PID возвращается родительскому процессу через возвращаемое значение `clone`.

Системный вызов clone способен и фактически используется для эмуляции классических системных вызовов fork(2) и vfork(2). Более новые версии glibc в случае ядра 2.6 используют clone для реализации системных вызовов fork(2) и vfork(2).

5.2.5. Блокировка

Блокировка реализована на уровне подсистем, поскольку не ожидается высокой конкуренции за эти ресурсы. Существует две блокировки: emul_lock, используемая для защиты манипуляций с linux_emuldata, и emul_shared_lock, используемая для манипуляций с linux_emuldata_shared. emul_lock представляет собой неспящий блокирующий мьютекс, в то время как emul_shared_lock — это спящий блокирующий sx_lock. Благодаря блокировке на уровне подсистем мы можем объединять некоторые блокировки, поэтому em_find предлагает доступ без блокировки.

5.3. TLS

Этот раздел посвящён TLS, также известному как локальное хранилище потока.

5.3.1. Введение в многопоточность

В компьютерных науках потоки (threads) — это сущности внутри процесса, которые могут планироваться независимо друг от друга. Потоки в процессе разделяют общие данные процесса (например, файловые дескрипторы), но также имеют свой собственный стек для своих данных. Иногда возникает необходимость в данных, специфичных для конкретного потока, но доступных на уровне процесса. Например, имя выполняемого потока или что-то подобное. Традиционный API для работы с потоками в UNIX® — pthreads — предоставляет способ сделать это через функции pthread_key_create(3), pthread_setspecific(3) и pthread_getspecific(3), где поток может создать ключ к локальным данным потока и использовать pthread_getspecific(3) или pthread_getspecific(3) для управления этими данными. Легко заметить, что это не самый удобный способ. Поэтому различные разработчики компиляторов C/C++ предложили более удобный метод. Они ввели новое ключевое слово thread, которое указывает, что переменная является специфичной для потока. Также был разработан новый метод доступа к таким переменным (по крайней мере, на архитектуре i386). Метод pthreads обычно реализуется в пользовательском пространстве в виде простой таблицы поиска. Производительность такого решения не очень высока. Новый метод использует (на i386) сегментные регистры для адресации области, где хранится TLS (Thread-Local Storage), так что фактический доступ к переменной потока сводится к добавлению сегментного регистра к адресу, таким образом обращаясь через него. Сегментные регистры, обычно %gs и %fs, действуют как селекторы сегментов. Каждый поток имеет свою собственную область, где хранятся локальные данные потока, и сегмент должен загружаться при каждом переключении контекста. Этот метод очень быстрый и используется практически повсеместно в мире UNIX® на архитектуре i386. И FreeBSD, и Linux® реализуют этот подход, и он даёт очень хорошие результаты. Единственный недостаток — необходимость перезагружать сегмент при каждом переключении контекста, что может замедлять переключения. FreeBSD пытается минимизировать эти накладные расходы, используя только 1 дескриптор сегмента, в то время как Linux® использует 3. Интересно, что почти ничто не использует больше 1 дескриптора (только Wine, кажется, использует 2), поэтому Linux® платит эту необязательную цену при переключении

контекстов.

5.3.2. Сегменты на i386

Архитектура i386 реализует так называемые сегменты. Сегмент — это описание области памяти. Он включает базовый адрес (начало) области памяти, её конец (границу), тип, защиту и т.д. Доступ к памяти, описываемой сегментом, может осуществляться с использованием регистров селекторов сегментов (`%cs`, `%ds`, `%ss`, `%es`, `%fs`, `%gs`). Например, предположим, что у нас есть сегмент с базовым адресом 0x1234 и длиной, а также следующий код:

```
mov %edx,%gs:0x10
```

Это загрузит содержимое регистра `%edx` в ячейку памяти по адресу 0x1244. Некоторые сегментные регистры имеют специальное назначение, например, `%cs` используется для сегмента кода, а `%ss` — для сегмента стека, но `%fs` и `%gs` обычно не используются. Сегменты хранятся либо в глобальной таблице GDT, либо в локальной таблице LDT. Доступ к LDT осуществляется через запись в GDT. LDT может хранить больше типов сегментов. LDT может быть отдельной для каждого процесса. Обе таблицы определяют до 8191 записей.

5.3.3. Реализация на Linux® i386

Существует два основных способа настройки TLS в Linux®. Он может быть настроен при клонировании процесса с использованием системного вызова `clone` или с помощью вызова `set_thread_area`. Когда процесс передаёт флаг `CLONE_SETTLS` в `clone`, ядро ожидает, что память, на которую указывает регистр `%esi`, будет содержать пользовательское представление сегмента в Linux®, которое преобразуется в машинное представление сегмента и загружается в слот GDT. Слот GDT может быть указан номером или можно использовать -1, что означает, что система сама должна выбрать первый свободный слот. На практике подавляющее большинство программ используют только одну запись TLS и не заботятся о номере записи. Мы используем это в эмуляции и фактически зависим от этого.

5.3.4. Эмуляция Linux® TLS

i386

Загрузка TLS для текущего потока происходит путем вызова `set_thread_area`, тогда как загрузка TLS для второго процесса в `clone` выполняется в отдельном блоке в `clone`. Эти две функции очень похожи. Единственное различие заключается в фактической загрузке сегмента GDT, которая происходит при следующем переключении контекста для вновь созданного процесса, в то время как `set_thread_area` должен загрузить его напрямую. Код в основном делает следующее. Он копирует дескриптор сегмента в формате Linux® из пользовательского пространства. Код проверяет номер дескриптора, но поскольку он различается между FreeBSD и Linux®, мы немного имитируем его. Мы поддерживаем только индексы 6, 3 и -1. Число 6 — это оригинальный номер Linux®, 3 — оригинальный номер FreeBSD, а -1 означает авто-выбор. Затем мы устанавливаем номер дескриптора на константу 3 и копируем его обратно в пользовательское пространство. Мы полагаемся на

то, что процесс в пользовательском пространстве использует номер из дескриптора, но это работает в большинстве случаев (никогда не встречалось ситуации, когда это не срабатывало), так как процесс в пользовательском пространстве обычно передаёт 1. Затем мы преобразуем дескриптор из формата Linux® в машинозависимую форму (т.е. независимую от операционной системы) и копируем его в дескриптор сегмента, определённый FreeBSD. Наконец, мы можем загрузить его. Мы назначаем дескриптор PCB потока (блок управления процессом) и загружаем сегмент `%gs` с помощью `load_gs`. Эта загрузка должна выполняться в критической секции, чтобы ничто не могло нас прервать. Случай `CLONE_SETTLS` работает точно так же, только загрузка с помощью `load_gs` не выполняется. Сегмент, используемый для этого (сегмент номер 3), разделяется между процессами FreeBSD и Linux®, поэтому слой эмуляции Linux® не добавляет накладных расходов по сравнению с обычным FreeBSD.

amd64

Реализация amd64 аналогична реализации i386, но изначально не использовался 32-битный дескриптор сегмента для этой цели (поэтому даже нативные пользователи 32-битного TLS не работали), поэтому нам пришлось добавить такой сегмент и реализовать его загрузку при каждом переключении контекста (когда установлен флаг, сигнализирующий о использовании 32-битного режима). Кроме этого, загрузка TLS точно такая же, только номера сегментов отличаются, а формат дескриптора и загрузка немного различаются.

5.4. Фьютексы

5.4.1. Введение в синхронизацию

Потокам требуется некоторая синхронизация, и POSIX® предоставляет несколько её видов: мьютексы для взаимного исключения, блокировки чтения-записи для взаимного исключения с преобладанием операций чтения над записями и условные переменные для сигнализации об изменении состояния. Интересно отметить, что в API потоков POSIX® отсутствует поддержка семафоров. Реализации этих механизмов синхронизации сильно зависят от типа поддержки потоков, которая у нас есть. В чистой модели 1:M (пользовательское пространство) реализация может быть выполнена исключительно в пользовательском пространстве и, следовательно, быть очень быстрой (условные переменные, вероятно, будут реализованы с использованием сигналов, т.е. не быстро) и простой. В модели 1:1 ситуация также довольно ясна — потоки должны синхронизироваться с использованием средств ядра (что очень медленно, поскольку требуется выполнение системного вызова). Смешанный сценарий M:N просто комбинирует первый и второй подходы или полагается исключительно на ядро. Синхронизация потоков является важной частью программирования с использованием потоков, и её производительность может значительно влиять на итоговую программу. Недавние тесты в операционной системе FreeBSD показали, что улучшенная реализация `sx_lock` дала 40% прироста скорости в *ZFS* (где активно используются блокировки sx), это внутренние механизмы ядра, но это наглядно демонстрирует, насколько важна производительность примитивов синхронизации.

Многопоточные программы должны быть написаны с минимальной конкуренцией за блокировки. В противном случае, вместо выполнения полезной работы поток просто

ожидает блокировку. В результате, наиболее хорошо написанные многопоточные программы демонстрируют низкую конкуренцию за блокировки.

5.4.2. Введение в фьютексы

Linux® реализует 1:1 потоковую модель, то есть использует примитивы синхронизации в ядре. Как упоминалось ранее, хорошо написанные многопоточные программы имеют низкую конкуренцию за блокировки. Таким образом, типичная последовательность может выполняться как два атомарных увеличения/уменьшения счётчика ссылок мьютекса, что очень быстро, как показано в следующем примере:

```
pthread_mutex_lock(&mutex);
...
pthread_mutex_unlock(&mutex);
```

1:1 threading вынуждает нас выполнять два системных вызова для этих вызовов мьютекса, что очень медленно.

Решение, реализованное в Linux® 2.6, называется фьютексы. Фьютексы выполняют проверку на конкуренцию в пользовательском пространстве и вызывают примитивы ядра только в случае конкуренции. Таким образом, типичный случай обходится без вмешательства ядра. Это обеспечивает достаточно быструю и гибкую реализацию примитивов синхронизации.

5.4.3. API фьютексов

Системный вызов futex выглядит следующим образом:

```
int futex(void *uaddr, int op, int val, struct timespec *timeout, void *uaddr2, int
val3);
```

В этом примере `uaddr` — это адрес мьютекса в пользовательском пространстве, `op` — операция, которую мы собираемся выполнить, а остальные параметры имеют значение, зависящее от конкретной операции.

Фьютексы реализуют следующие операции:

- `FUTEX_WAIT`
- `FUTEX_WAKE`
- `FUTEX_FD`
- `FUTEX_REQUEUE`
- `FUTEX_CMP_REQUEUE`
- `FUTEX_WAKE_OP`

FUTEX_WAIT

Эта операция проверяет, что по адресу `uaddr` записано значение `val`. Если нет, возвращается `EWOULDBLOCK`, в противном случае поток ставится в очередь на фьютекс и приостанавливается. Если аргумент `timeout` не равен нулю, он задает максимальное время ожидания, в противном случае ожидание бесконечно.

FUTEX_WAKE

Эта операция захватывает фьютекс по адресу `uaddr` и пробуждает первые `val` фьютексов, ожидающих в очереди на этом фьютексе.

FUTEX_FD

Эта операция связывает файловый дескриптор с заданным фьютексом.

FUTEX_REQUEUE

Эта операция берет `val` потоков, ожидающих на фьютексе по адресу `uaddr`, пробуждает их и берет следующие `val2` потоков, перемещая их в очередь фьютекса по адресу `uaddr2`.

FUTEX_CMP_REQUEUE

Эта операция делает то же самое, что и `FUTEX_REQUEUE`, но сначала проверяет, что `val3` равно `val`.

FUTEX_WAKE_OP

Эта операция выполняет атомарную операцию над `val3` (которая содержит закодированное другое значение) и `uaddr`. Затем она пробуждает `val` потоков на фьютексе по адресу `uaddr`, и если атомарная операция вернула положительное число, пробуждает `val2` потоков на фьютексе по адресу `uaddr2`.

Операции, реализованные в `FUTEX_WAKE_OP`:

- `FUTEX_OP_SET`
- `FUTEX_OP_ADD`
- `FUTEX_OP_OR`
- `FUTEX_OP_AND`
- `FUTEX_OP_XOR`

В прототипе системного вызова futex отсутствует параметр `val2`. Значение `val2` берётся из параметра `struct timespec *timeout` для операций `FUTEX_REQUEUE`, `FUTEX_CMP_REQUEUE` и `FUTEX_WAKE_OP`.

5.4.4. Эмуляция фьютексов в FreeBSD

Эмуляция futex в FreeBSD взята из NetBSD и дополнительно расширена нами. Она размещена в файлах `linux_futex.c` и linux_futex.h. Структура `futex` выглядит следующим

образом:

```
struct futex {
  void *f_uaddr;
  int f_refcount;

  LIST_ENTRY(futex) f_list;

  TAILQ_HEAD(lf_waiting_paroc, waiting_proc) f_waiting_proc;
};
```

И структура `waiting_proc` выглядит следующим образом:

```
struct waiting_proc {

  struct thread *wp_t;

  struct futex *wp_new_futex;

  TAILQ_ENTRY(waiting_proc) wp_list;
};
```

futex_get / futex_put

Фьютекс получается с помощью функции `futex_get`, которая выполняет поиск в линейном списке фьютексов и возвращает найденный или создаёт новый. При освобождении фьютекса после использования вызывается функция `futex_put`, которая уменьшает счетчик ссылок фьютекса, и если счетчик достигает нуля, фьютекс освобождается.

futex_sleep

Когда фьютекс ставит поток в очередь на ожидание, он создаёт структуру `working_proc` и помещает эту структуру в список внутри структуры futex, после чего просто выполняет tsleep(9) для приостановки потока. Ожидание может быть ограничено по времени. После возврата из tsleep(9) (поток был разбужен или истекло время ожидания) структура `working_proc` удаляется из списка и уничтожается. Все это выполняется в функции `futex_sleep`. Если мы были разбужены с помощью `futex_wake`, у нас установлен `wp_new_futex`, поэтому мы ожидаем на нём. Таким образом, фактическое перемещение выполняется в этой функции.

futex_wake

Пробуждение потока, ожидающего на фьютексе, выполняется в функции `futex_wake`. Сначала в этой функции мы имитируем странное поведение Linux®, где пробуждаются N потоков для всех операций, за исключением того, что операции REQUEUE выполняются на N+1 потоках. Однако обычно это не имеет значения, так как мы пробуждаем все потоки. Далее в функции в цикле мы пробуждаем n потоков, после чего проверяем, есть ли новый фьютекс для перестановки. Если есть, мы переставляем до n2 потоков на новый futex. Это

взаимодействует с `futex_sleep`.

futex_wake_op

Операция `FUTEX_WAKE_OP` довольно сложна. Сначала мы получаем два фьютекса по адресам `uaddr` и `uaddr2`, затем выполняем атомарную операцию с использованием `val3` и `uaddr2`. После этого пробуждаются `val` ожидающих на первом фьютексе, и если условие атомарной операции выполняется, мы пробуждаем `val2` (т.е. `timeout`) ожидающих на втором фьютексе.

Атомарная операция на фьютексе

Атомарная операция принимает два параметра `encoded_op` и `uaddr`. Закодированная операция включает саму операцию, сравниваемое значение, аргумент операции и аргумент сравнения. Псевдокод операции выглядит следующим образом:

```
oldval = *uaddr2
*uaddr2 = oldval OP oparg
```

И это выполняется атомарно. Сначала происходит копирование числа по адресу `uaddr`, а затем выполняется операция. Код обрабатывает ошибки страниц, и если ошибки не происходит, `oldval` сравнивается с аргументом `cmparg` с помощью компаратора cmp.

Блокировка фьютекса

Реализация фьютексов использует два списка блокировок для защиты `sx_lock` и глобальных блокировок (либо Giant, либо другой `sx_lock`). Каждая операция выполняется заблокированной от начала до самого конца.

5.5. Реализация различных системных вызовов

В этом разделе я опишу несколько менее значимых системных вызовов, которые стоит упомянуть, потому что их реализация неочевидна или эти вызовы представляют интерес с другой точки зрения.

5.5.1. *семейство системных вызовов at

Во время разработки ядра Linux® 2.6.16 были добавлены *at-системные вызовы. Эти системные вызовы (например, `openat`) работают точно так же, как их аналоги без at, за исключением параметра `dirfd`. Этот параметр определяет местоположение файла, над которым выполняется системный вызов. Если параметр `filename` является абсолютным, `dirfd` игнорируется, но если путь к файлу относительный, `dirfd` вступает в игру. Параметр `dirfd` представляет собой каталог, относительно которого проверяется относительный путь. Параметр `dirfd` является файловым дескриптором некоторого каталога или `AT_FDCWD`. Например, системный вызов `openat` может выглядеть следующим образом:

```
file descriptor 123 = /tmp/foo/, current working directory = /tmp/
```

```
openat(123, /tmp/bah\, flags, mode) /* opens /tmp/bah */
openat(123, bah\, flags, mode)      /* opens /tmp/foo/bah */
openat(AT_FDWCWD, bah\, flags, mode)    /* opens /tmp/bah */
openat(stdio, bah\, flags, mode)    /* returns error because stdio is not a directory
*/
```

Эта инфраструктура необходима для избежания состояний гонки при открытии файлов вне рабочего каталога. Представьте, что процесс состоит из двух потоков, потока A и потока B. Поток A выполняет `open(./tmp/foo/bah., flags, mode)`, и перед возвратом управления он вытесняется, и начинает выполняться поток B. Поток B не учитывает потребности потока A и переименовывает или удаляет /tmp/foo/. Возникает состояние гонки. Чтобы избежать этого, мы можем открыть /tmp/foo и использовать его как `dirfd` для системного вызова `openat`. Это также позволяет пользователю реализовать рабочие каталоги для каждого потока.

Семейство *at системных вызовов Linux® включает: `linux_openat`, `linux_mkdirat`, `linux_mknodat`, `linux_fchownat`, `linux_futimesat`, `linux_fstatat64`, `linux_unlinkat`, `linux_renameat`, `linux_linkat`, `linux_symlinkat`, `linux_readlinkat`, `linux_fchmodat` и `linux_faccessat`. Все они реализованы с использованием модифицированной функции namei(9) и простого слоя обёртки.

Реализация

Реализация выполнена путем изменения функции namei(9) (описанной выше) для приема дополнительного параметра `dirfd` в структуре `nameidata`, который указывает начальную точку для поиска пути вместо использования текущего рабочего каталога каждый раз. Преобразование `dirfd` из номера файлового дескриптора в vnode выполняется в нативных *at-системных вызовах. Когда `dirfd` равен `AT_FDCWD`, запись `dvp` в структуре `nameidata` имеет значение `NULL`, но если `dirfd` представляет другой номер, мы получаем файл по этому дескриптору, проверяем его валидность и, если к нему прикреплен vnode, получаем этот vnode. Затем проверяем, является ли этот vnode каталогом. В самой функции namei(9) мы просто заменяем vnode `dvp` на переменную `dp` в функции namei(9), которая определяет начальную точку. Функция namei(9) используется не напрямую, а через цепочку различных функций на разных уровнях. Например, `openat` работает следующим образом:

```
openat() --> kern_openat() --> vn_open() -> namei()
```

По этой причине `kern_open` и `vn_open` должны быть изменены для включения дополнительного параметра `dirfd`. Слой совместимости для них не создаётся, так как пользователей этих функций немного и их можно легко адаптировать. Данная общая реализация позволяет FreeBSD реализовать свои собственные *at-системные вызовы. Это обсуждается в настоящее время.

5.5.2. Ioctl

Интерфейс ioctl довольно хрупок из-за своей обобщённости. Необходимо учитывать, что устройства в Linux® и FreeBSD различаются, поэтому требуется особая осторожность для корректной работы эмуляции ioctl. Обработка ioctl реализована в файле linux_ioctl.c, где

определена функция `linux_ioctl`. Эта функция просто перебирает наборы обработчиков ioctl, чтобы найти обработчик, реализующий данную команду. Системный вызов ioctl имеет три параметра: файловый дескриптор, команду и аргумент. Команда представляет собой 16-битное число, которое теоретически делится на старшие 8 бит, определяющие класс команды ioctl, и младшие 8 бит, которые являются конкретной командой в данном наборе. Эмуляция использует это разделение. Реализованы обработчики для каждого набора, такие как `sound_handler` или `disk_handler`. Каждый обработчик имеет определённые максимальную и минимальную команды, которые используются для выбора нужного обработчика. Существуют небольшие проблемы с этим подходом, поскольку Linux® не всегда последовательно использует разделение на наборы, поэтому иногда ioctls для другого набора оказываются внутри набора, к которому они не должны принадлежать (например, SCSI generic ioctls внутри набора cdrom и т.д.). В настоящее время FreeBSD реализует не так много ioctls Linux® (по сравнению с NetBSD, например), но планируется перенести их из NetBSD. Тенденция такова, что ioctls Linux® используются даже в родных драйверах FreeBSD для упрощения портирования приложений.

5.5.3. Отладка

Каждый системный вызов должен поддерживать отладку. Для этой цели мы вводим небольшую инфраструктуру. У нас есть механизм `ldebug`, который определяет, нужно ли отлаживать данный системный вызов (настраивается через `sysctl`). Для вывода сообщений используются макросы `LMSG` и `ARGS`. Они применяются для форматирования строк вывода с целью создания единообразных отладочных сообщений.

6. Заключение

6.1. Результаты

По состоянию на апрель 2007 года уровень эмуляции Linux® способен достаточно хорошо эмулировать ядро Linux® 2.6.16. Оставшиеся проблемы касаются фьютексов, незавершённого семейства системных вызовов *at, проблематичной доставки сигналов, отсутствия `epoll` и `inotify`, а также, вероятно, некоторых ошибок, которые мы ещё не обнаружили. Несмотря на это, мы способны запускать практически все программы Linux®, включённые в Коллекцию портов FreeBSD, с Fedora Core 4 на ядре 2.6.16, а также есть некоторые предварительные сообщения об успешной работе с Fedora Core 6 на ядре 2.6.16. Недавно был добавлен linux_base Fedora Core 6, что позволило провести дополнительные тестирования уровня эмуляции и дало нам больше подсказок, куда следует направить усилия для реализации недостающих функций.

Мы можем запускать наиболее популярные приложения, такие как www/linux-firefox, net-im/skype, и некоторые игры из Коллекции портов. Некоторые программы демонстрируют некорректное поведение при эмуляции 2.6, но это в настоящее время исследуется, и, надеемся, скоро будет исправлено. Единственное крупное приложение, которое, как известно, не работает, — это Linux® Java™ Development Kit, и это связано с требованием функции `epoll`, которая не имеет прямого отношения к ядру Linux® 2.6.

Мы надеемся включить эмуляцию 2.6.16 по умолчанию через некоторое время после выхода

FreeBSD 7.0, по крайней мере, чтобы открыть части эмуляции 2.6 для более широкого тестирования. Как только это будет сделано, мы сможем перейти на Fedora Core 6 linux_base, что является конечной целью.

6.2. Будущие работы

Будущая работа должна быть сосредоточена на исправлении оставшихся проблем с фьютексами, реализации оставшихся системных вызовов семейства *at, исправлении доставки сигналов и, возможно, реализации механизмов `epoll` и `inotify`.

Мы надеемся вскоре добиться безупречной работы наиболее важных программ, чтобы можно было по умолчанию переключиться на эмуляцию 2.6 и сделать Fedora Core 6 базовой версией linux_base, поскольку используемая в настоящее время Fedora Core 4 больше не поддерживается.

Другая возможная цель — поделиться нашим кодом с NetBSD и DragonflyBSD. NetBSD имеет некоторую поддержку эмуляции 2.6, но она далека от завершения и не была должным образом протестирована. DragonflyBSD выразила некоторую заинтересованность в переносе улучшений версии 2.6.

В целом, по мере развития Linux® мы хотели бы идти в ногу с их разработкой, реализуя новые системные вызовы. В первую очередь на ум приходит `splice`. Некоторые уже реализованные системные вызовы также неоптимальны, например `mremap` и другие. Также можно внести некоторые улучшения производительности, такие как более детальная блокировка и другие.

6.3. Команда

Я сотрудничал в этом проекте с (в алфавитном порядке):

- John Baldwin <jhb@FreeBSD.org>
- Konstantin Belousov <kib@FreeBSD.org>
- Emmanuel Dreyfus
- Scot Hetzel
- Jung-uk Kim <jkim@FreeBSD.org>
- Alexander Leidinger <netchild@FreeBSD.org>
- Suleiman Souhlal <ssouhlal@FreeBSD.org>
- Li Xiao
- David Xu <davidxu@FreeBSD.org>

Я хотел бы поблагодарить всех этих людей за их советы, рецензирование кода и общую поддержку.

7. Литература

1. Marshall Kirk McKusick - George V. Neville-Neil. Design and Implementation of the FreeBSD operating system. Addison-Wesley, 2005 год.
2. https://tldp.org
3. https://www.kernel.org