Апрель 2006

Оригинальный материал: http://www.usenix.com/publications/login/2006-04/openpdfs/herder.pdf

Перевод: Павел Макаров (makarov@minix3.ru)

Модульное системное программирование в MINIX 3

Йоррит Н. Эрдер (Jorrit N. Herder), Герберт Бос (Herbert Bos), Бен Грас (Ben Gras), Филип Хомбург (Philip Homburg) и Эндрю С. Таненбаум (Andrew S. Tanenbaum)

Университет Врийе, Амстердам

Когда в начале 1960-х первые современные операционные системы только разрабатывались, разработчики были так озабочены их производительностью, что писали их на ассемблере даже несмотря на то, что такие высокоуровневые языки, как FORTRAN, MAD и Algol были уже широко признаны. Надёжность и безопасность вообще не попали в поле зрения разработчиков. Но времена изменились и сегодня нам следует пересмотреть наше отношение к надёжности операционных систем.

Если вы спросите обычных компьютерных пользователей, что они ценят менее всего в своей нынешней операционной системе, только малая их часть упомянет скорость. В остальном это будет примерно равное количество сетований на чрезмерную сложность, недостаток надёжности и безопасности в широком понимании (вирусы, черви и т.п.). Мы уверены, что большая часть этих проблем произрастает из проектных решений 40- или 50-летней давности. В частности, стремление первых поколений разработчиков поставить быстродействие превыше всего остального привела к монолитным конструкциям со всей операционной системой, работающей как единая исполняемая программа в режиме ядра. В те времена, когда максимальный объём памяти для операционной системы был только 32K слов, как было в случае с CTSS - первой ОС с разделением времени Массачусетского Технологического Института (МТИ), операционные системы со многими миллионами строк кода были просто невозможны и поэтому сложность ОС была поддающейся осмыслению и контролю.

Допустимый размер памяти рос, менялись и операционные системы, пока мы не получили нынешние операционные системы с сотнями функций, взаимодействующих таким сложным способом, что на самом деле никто не может понять вообще, как они работают. И если Windows XP с её 5 миллионами строк кода ядра является наихудшим примером с этой точки зрения, то Linux с его 3 миллионами строк кода ядра быстро продвигается по тому же пути. Мы думаем, этот путь ведёт в тупик.

Различные исследования показали, что допустимое количество ошибок в программах должно лежать в диапазоне 1÷20 на 1000 строк кода[1]. Более того, операционные системы обнаруживают тенденцию быть сложнее и изощрённее, чем прикладные программы, а драйверы устройств имеют на порядок большее количество ошибок на тысячу строк кода, чем вся остальная часть операционной системы[2,3]. Позволяя миллионам строк малопонятного кода взаимодействовать таинственным образом в пределах единого адресного пространства, не стоит удивляться, что мы имеем проблемы с надёжностью и безопасностью.

Надёжность операционной системы

С нашей точки зрения, единственным путём улучшения надёжности операционной системы является отказ от модели ОС как гигантской программы, работающей в режиме ядра, когда любая строка кода способна подвергнуть риску или вообще «уронить» систему. Почти вся функциональность ОС, и особенно все драйверы устройств, должны быть переведены в разряд процессов пользователя, оставляя только крошечное микроядро работающим в режиме ядра. Перемещение всей ОС в единый процесс пространства пользователя, как это было сделано в L4Linux[4], делает перезагрузку ОС после сбоя быстрее, но не решает фундаментальную проблему потенциальной критической опасности каждой строки кода. Всё, что для этого требуется – это разделить базовую функциональность ОС, включая файловую систему, управление процессами и графику, на множество процессов, поместив каждый драйвер устройства в отдельный процесс и очень жёстко контролируя, что каждый компонент может делать. Только на основе такой архитектуры у нас на самом деле есть шанс улучшить надёжность системы.

Причин, по которым такая модульная, мультисерверная конструкция лучше, чем монолитная, три. Во-первых, помещая большую часть кода из пространства ядра в пространство

пользователя, мы не уменьшаем количество ошибок, но мы уменьшаем возможность каждой ошибки вызвать сбой. Ошибки в процессах из пространства пользователя имеют гораздо меньше возможности испортить критические структуры данных ядра и не могут затронуть оборудование, которого они не должны касаться при нормальной работе. Сбой процесса из пространства пользователя редко бывает фатальным, в то время как сбой в ядре фатален всегда. Выводя большую часть кода из ядра, мы также выводим оттуда и большую часть ошибок.

Во-вторых, разбивая ОС на множество процессов, каждый из которых работает в своём собственном адресном пространстве, мы значительно ограничиваем распространение сбоев. Ошибка в аудиодрайвере может выключить звук, но она не может случайно уничтожить файловую систему. В ОС с монолитным ядром, наоборот, ошибки в любой функции могут разрушить код и структуры данных в других, не связанных с этой функцией и гораздо более критических функциях.

В-третьих, конструируя систему как набор пользовательских процессов, функциональность каждого модуля может быть ясно определена, что делает всю систему гораздо более лёгкой для понимания и простой для применения. Кроме того, удобство сопровождения ОС будет улучшено, поскольку модули могут обслуживаться независимо друг от друга до тех пор, пока сохраняются интерфейсы и разделяемые структуры данных.

Поскольку эта статья не нацелена непосредственно на безопасность, важно отметить, что надёжность и безопасность ОС тесно связаны. Безопасность обычно закладывается в модели многопользовательских ОС, а не в однопользовательские, в которых этот пользователь может исполнять посторонний (или «враждебный» - hostile) код. Однако, множество проблем безопасности вызывается злонамеренным кодом, внедрённым с помощью вирусов или червей через ошибки, такие, как переполнение буферов. Выводя большую часть кода из ядра, воздействие компонентов ОС можно сделать существенно менее мощным. Несанкционированная запись стека аудиодрайвера может позволить незваному гостю заставить компьютер издавать таинственные шумы, но не может скомпрометировать безопасность системы, поскольку аудиодрайвер не имеет привилегий суперпользователя. Таким образом, поскольку мы не будем более в этой статье обсуждать безопасность, то отметим только, что наша конструкция имеет также огромный потенциал по улучшению безопасности ОС.

Утверждение, что микроядра хороши с точки зрения надёжности, не ново. В 1980-х и 1990-х годах были разработаны многочисленные микроядра, включая L4[5], Mach[6], V[7], Chorus[8] и Amoeba[9]. Никто из них не смог заменить монолитные ОС на микроядерные операционные системы, однако мы многому научились с тех пор и пришло время попытаться сделать это ещё раз. Даже Microsoft понимает это. Следующая версия Windows (Vista) будет включать множество драйверов, работающих в режиме пользователя, да и исследовательский проект Microsoft Singularity также основан на микроядре.

Архитектура MINIX 3

Для проверки наших идей мы разработали POSIX-совместимую экспериментальную систему. В качестве основы для макета мы использовали ОС MINIX по причине её чрезвычайно маленького размера и длинной истории развития. MINIX – это свободная микроядерная операционная система, которая доступна с полным исходным кодом, в большинстве своём написанном на C. Первоначальная версия была написана одним из авторов (Энди Таненбаумом) в 1987 году и была изучена многими десятками тысяч студентов в сотнях университетов за 19 лет; за последние 10 лет практически вообще не было сообщений об ошибках в ядре, по-видимому, из-за его маленького размера.

Мы начали с MINIX 2 и затем очень сильно модифицировали её, выведя драйверы из ядра и т.п., но мы решили сохранить имя и назвали новую систему MINIX 3. Она основана на микроядре, содержащем сегодня не более 4'000 строк кода, с различными серверами и драйверами в пространстве пользователя, что в совокупности образовало операционную систему, показанную на рис. 1. Несмотря на эту нетрадиционную структуру, пользователю система покажется ещё одним вариантом UNIX. Она поддерживает два компилятора C (ACK и gcc), также как и множество популярных утилит – Emacs, vi, Perl, Python, Telnet, FTP и 300 других (*более 400 на сентябрь 2006 года – прим. переводчика*). Ранее в неё также была портирована и система X Windows. MINIX 3 доступна на http://www.minix3.org со всеми исходными кодами и под лицензией BSD.

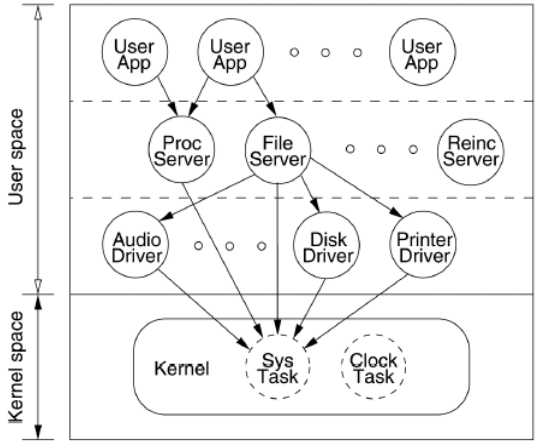

Рис. 1. Изображение многоуровневой архитектуры MINIX 3

Все приложения, серверы и драйверы запускаются как изолированные процессы в пространстве пользователя. Крошечное выверенное ядро является единственной частью ОС, которая запускается в режиме ядра. Расслоение является логическим, поскольку все пользовательские процессы обрабатываются ядром одинаково.

Вкратце, микроядро поддерживает аппаратные прерывания, низкоуровневое управление памятью, планирование процессов и межпроцессное взаимодействие. Последнее выполняется с помощью примитивов, позволяющих процессам посылать сообщения фиксированной длины в адрес других процессов, которым они имеют право их (эти сообщения) посылать. Большая часть взаимодействия является синхронной, при этом осуществляется блокирование отправителя или получателя сообщений в случае, если другой абонент не готов. Отправка сообщения занимает примерно 500 нс на 2,2 МГц процессоре Athlon. Хотя системный вызов обычно состоит из двух сообщений (запроса и ответа), даже 10'000 системных вызовов в секунду будут использовать только 1% CPU, так что издержки передачи сообщений практически вообще не влияют на производительность. Кроме того, существует неблокируемый механизм уведомления о событии (nonblocking event notification mechanism). Незавершённые уведомления запоминаются в компактной битовой карте, которая статически объявлена как часть таблицы процесса. Такая схема прохождения сообщений исключает все проблемы управления и переполнения буфера ядра, также, как и множество взаимоблокировок (deadlocks).

Следующий верхний уровень содержит драйверы устройств, по одному на основное устройство. Каждый драйвер является пользовательским процессом, защищённым диспетчером памяти (MMU) таким же образом, как и обычный процесс пользователя. Они (драйверы) отличаются только тем, что им позволено осуществлять небольшое число системных вызовов для получения обслуживания со стороны ядра (kernel services). Типичными вызовами ядра являются запись набора значений в аппаратный порт ввода-вывода или запрос о том, что данные будут скопированы в или из пользовательского процесса. Битовая карта в таблице процесса ядра отслеживает, какие вызовы каждый драйвер (или сервер) может делать. Ядро также знает, какие порты ввода-вывода драйверу позволено использовать; копирование оказывается возможным только с явного разрешения.

Интерфейс операционной системы формируется набором серверов. Основные из них – это *файловый сервер*, *диспетчер процессов* (process manager) и *сервер реинкарнации*. Пользовательские процессы осуществляют POSIX-образные системные вызовы посылкой сообщения одному из этих серверов, который затем выполняет вызов. Сервер реинкарнации представляет особый интерес, поскольку он является родительским процессом всех серверов и драйверов. Он отличается от *init*, от которого порождаются обычные пользовательские процессы, поскольку он управляет операционной системой и защищает её. Если сервер или драйвер отказывает или завершается иным способом, он становится «зомби» до тех пор, пока сервер реинкарнации не отловит его во время просмотра его таблиц для определения того, что следует делать. Обычной реакцией является создание нового драйвера или сервера и информирование остальных процессов о том, что было сделано.

Наконец, у нас есть обычные пользовательские процессы, которые обладают способностью

посылать сообщения фиксированной длины некоторым серверам для запроса услуг, но обычно не имеют никаких других полномочий. И если прохождение такого сообщения протекает скрытно, то системные библиотеки предоставляют программисту нормальный POSIX API.

Жизнь с программными ограничениями

Объяснив, почему нужны микроядра и как структурирована MINIX 3, настало самое время заняться самым главным в этой статье: моделью программирования в MINIX 3 и её применением. Ниже мы укажем некоторые отличительные свойства, сильные и слабые стороны модели программирования. Но прежде, чем мы начнём, полезно вспомнить истории о том, как ограничение программистов в том, что они могут делать, часто приводило к более надёжному коду. Рассмотрим несколько примеров.

Во-первых, после появления первых диспетчеров памяти (MMU) пользовательские программы для осуществления операций ввода-вывода были вынуждены выполнять системные вызовы, а не просто запускать устройства ввода-вывода напрямую. Естественно, некоторые из пользователей выражали недовольство тем, что вызовы ядра были медленнее, чем прямое общение с устройствами ввода-вывода (и они были правы). Но в конечном итоге все единодушно согласились, что это ограничение, на которое программист должен был пойти, обернулось преимуществом, благодаря которому за счёт небольшой потери быстродействия ошибки в пользовательской программе больше не приводили к аварийному отказу компьютера.

Во-вторых, когда Дейкстра (E.W. Dijkstra) написал своё знаменитое письмо о вреде операторов перехода GOTO «Goto Statement Considered Harmful»[10], громкие крики и стенания раздались со стороны многих программистов, которые считали, что возникла угроза их стилю написания витиеватых (spaghetti-like) программ. Несмотря на эти первоначальные возражения, идея всё же прижилась и программисты стали учиться писать хорошо структурированные программы.

В-третьих, когда было анонсировано объектно-ориентированное программирование, многие программисты игнорировали эту идею, поскольку они не могли бы больше рассчитывать на чтение или «вылизывание» структур данных внутри других объектов, что прежде являлось обычным делом во имя эффективности. Например, когда Java была представлена программистам C, многие из них усмотрели в ней аналог смирительной рубашки, поскольку они не смогли бы больше свободно манипулировать указателями. Тем не менее, объектно-ориентированное программирование сегодня повсеместно распространено и привело к созданию программного кода лучшего качества.

Ограничения MINIX 3

Аналогичным образом, модель программирования в MINIX 3 также является более ограничивающей по отношению к разработчикам операционной системы, чем было до неё, но мы считаем, что эти ограничения приведут исключительно к более надёжной системе. В настоящее время MINIX 3 написана на C, но постепенное переписывание некоторых её модулей на языке, обеспечивающем типовую безопасность (type-safe language), таком, например, как Cyclone, могло бы стать когда-нибудь возможным. Давайте начнём наш обзор модели, взглянув на некоторые из этих ограничений.

Ограниченный доступ к ядру. Ядро MINIX 3 экспортирует различные вызовы ядра для поддержки серверов и драйверов операционной системы, работающих в пространстве пользователя. Каждый драйвер и сервер имеет битовую карту в таблице процесса, ограничивающую вызовы ядра, которые он может использовать. Эта защита очень тонкая (quite fine-grained), так что, например, драйвер устройства может иметь право выполнять ввод-вывод или делать копии в и из пользовательских процессов, но не остановку системы, не создание новых процессов или установку/сброс собственно ограничений (restriction policies).

Защита памяти. В мультисерверной конструкции MINIX 3 все серверы и драйверы операционной системы запускаются как изолированные процессы пользователя. Каждый помещён в собственное адресное пространство, защищаемое аппаратным диспетчером памяти (MMU). Попытка нелегального доступа в память другого процесса приводит к нештатной ситуации MMU и вызывает немедленное уничтожение «нарушителя» диспетчером процессов. Естественно, файловой системе и драйверам устройств нужно взаимодействовать с пользовательскими процессами для выполнения ввода-вывода, но это делается через использование безопасных виртуальных копий при посредничестве ядра. Копирование другому процессу возможно только, когда есть ясное разрешение от этого процесса или от надёжного

процесса (trusted process), такого, как файловая система. Эта схема снимает ответственность с драйверов и предотвращает повреждение памяти.

Ограниченный доступ к порту ввода-вывода. Каждый драйвер имеет ограниченный диапазон портов ввода-вывода, к которым он может обращаться. Так как пользовательские процессы не имеют привилегий ввода-вывода, ядро всегда посредничает в этом и может проверить, был ли разрешён запрос на ввод-вывод. Разрешённые порты ввода-вывода устанавливаются, как только драйвер стартует. Для ISA устройств это делается с помощью конфигурационных файлов, для PCI устройств набор портов ввода-вывода автоматически задаётся сервером шины PCI. Корректный набор портов для каждого драйвера сохраняется в соответствующем месте таблицы процесса драйвера (driver's process table entry) в ядре. Такая защита гарантирует, что драйвер принтера не может случайно записать мусор на диск, потому что любая попытка записи в порт ввода-вывода диска будет приводить к неудачному вызову ядра. Серверы и обычные пользовательские процессы вообще не имеют доступа к какому бы то ни было порту ввода-вывода.

Ограниченное межпроцессное взаимодействие. Процессы не могут посылать сообщения к случайным процессам. Кроме того, ядро отслеживает, кто с кем может общаться, и предотвращает нарушения. Разрешённые функции и адресаты IPC устанавливаются сервером реинкарнации при запуске нового системного процесса. Например, драйверу может быть разрешено взаимодействовать только с файловым сервером и ни с каким другим процессом. Эта возможность устраняет некоторые ошибки, когда процесс пытается послать сообщение другому процессу, который его (сообщение) не ожидает.

Разработка операционной системы в пространстве пользователя

А сейчас давайте посмотрим на остальные аспекты модели программирования в MINIX 3. Несмотря на существование вышеупомянутых ограничений, мы полагаем, что программирование в среде мультисерверной операционной системы имеет множество преимуществ и может привести к более высокой производительности и лучшему качеству кода.

Короткий цикл разработки. Огромная разница между монолитной и мультисерверной ОС сразу же становится ясной, как только вы обратите внимание на цикл разработки компонентов операционной системы. Системное программирование в монолитной ОС обычно включает редактирование, компиляцию, пересборку ядра и перезагрузку для проверки нового компонента. последующий сбой может потребовать ещё одной перезагрузки с последующей утомительной низкоуровневой отладкой, зачастую даже без дампа ядра. В противоположность этому, цикл разработки на мультисерверной ОС, такой, как MINIX 3, существенно короче. Обычно шаги ограничиваются редактированием, компиляцией, тестированием и отладкой. Далее мы детально рассмотрим эти шаги.

Обычная модель программирования. Поскольку драйверы и серверы являются просто обычными пользовательскими процессами, они могут использовать любые библиотеки, которые им нужны. В некоторых случаях могут быть использованы даже POSIX-образные системные вызовы. Возможность делать это может не соответствовать более жёсткой среде, доступной программистам, пишущим код для монолитных ядер. По существу, работа в пространстве пользователя делает программирование более лёгким.

Никакого вынужденного простоя системы. Неизбежные перезагрузки монолитных ОС практически добивают всех пользователей, вынуждая их предпочесть определённые инструментальные средства разработки. В MINIX 3 для тестирования новых компонентов никаких перезагрузок не требуется, так что остальных пользователей это никак не задевает. Более того, ошибки или другие проблемы изолированы в новом компоненте и не касаются системы в целом, поскольку новый компонент запускается как независимый процесс в ограниченной среде исполнения. Проблемы, таким образом, не могут распространяться, как в монолитном ядре.

Лёгкая отладка. Отладка драйвера устройства в монолитном ядре – это реальная проблема. Зачастую система сразу же останавливается и программист не получает ответа на вопрос о том, что пошло не так. Использование симулятора или эмулятора обычно бесполезно, поскольку, как правило, устройство, для которого пишется драйвер, очень новое и не поддерживается симулятором или эмулятором. В противоположность этому, в модели MINIX 3 драйвер устройства – это просто пользовательский процесс, так что, если он «рухнет», это произойдёт вне дампа ядра, который может быть проанализирован любым нормальным средством отладки.

В дополнение, выход всех операторов *printf()* в драйверах и серверах автоматически направляется в лог-сервер, который пишет всё это в файл. После неудачного запуска нового драйвера программист может просмотреть лог с тем, чтобы увидеть, что драйвер сделал перед тем, как «умереть».

Низкий барьер для доступа. Так как написание драйверов и серверов в MINIX 3 гораздо легче, чем в обычных ОС, то исследователи и другие не профессиональные программисты легко могут попытаться написать новые. Лёгкость экспериментирования может существенно развить систему, позволяя людям с хорошими идеями, но малым опытом системного программирования попытаться реализовать их идеи и строить макеты, которые они никогда бы не создали в монолитных ядрах. Хотя наитруднейшая часть написания нового драйвера устройства требует понимания реальной аппаратной части, остальные компоненты ОС могут быть реализованы совсем легко. Например, демонстрационный пример в конце этой статьи иллюстрирует, как в MINIX 3 могут быть добавлены семафоры.

Высокая производительность. Поскольку разработка ОС в пространстве пользователя легче, программист может выполнить работу быстрее. Кроме того, поскольку после удаления ошибки не требуется выполнять никаких длиннющих системных сборок (lengthy system build), то экономится дополнительное время. Наконец, так как систему не надо перезагружать после сбоя драйвера, то, как только программист проанализирует дамп ядра и лог и обновит код, он может протестировать новый драйвер без перезагрузки системы. В монолитном ядре обычно нужно выполнить две перезагрузки: одну для запуска системы после сбоя и ещё одну для загрузки заново собранного ядра.

Хорошая отслеживаемость. Когда драйвер или сервер «падает», то совершенно очевидно, что именно «упало» (так как его «родитель», сервер реинкарнации, знает, какой процесс завершил работу). Как следствие, гораздо легче, нежели в монолитных системах, отловить, чьей ошибкой был вызван сбой и, возможно, кто конкретно должен ответить за произошедшее. Так что, делая изготовителей коммерческого программного обеспечения ответственными за их ошибки абсолютно таким же способом, как и изготовителей шин, лекарств и другой контролируемой продукции, можно улучшить качество программного обеспечения.

Большая гибкость. Наша модульная модель предлагает большую гибкость и делает системное администрирование гораздо легче. Поскольку модули ОС – это всего лишь процессы, то их относительно просто заменять. Становится легче конфигурировать ОС, добавляя и настраивая отдельные модули. Более того, если необходимо исправить драйвер устройства, то обычно это можно сделать «на лету», без прекращения обслуживания или остановки системы. Замена модуля в монолитных ядрах гораздо труднее и часто требует перезагрузки. И, наконец, поддержка также становится легче, поскольку все модули малы, независимы и совершенно понятны.

Пример: управляемое сообщениями программирование в MINIX 3

Сейчас мы оценим модель программирования в MINIX 3 с помощью небольшого примера, который показывает, как в MINIX 3 могут быть добавлены семафоры. Хотя это проще, чем применение новой файловой системы или драйвера устройства, но это показывает некоторые важные аспекты MINIX 3.

Семафоры являются положительными целыми, равными или большими нуля, и поддерживают две операции – UP и DOWN, для синхронизации нескольких процессов, пытающихся получить доступ к разделяемому ресурсу. Операция DOWN на семафоре *S* уменьшает *S* до тех пор, пока *S* не станет равно нулю, и в этом случае он блокирует вызывающий процесс до тех пор, пока какой-нибудь не увеличит *S* с помощью операции UP. Такая функциональность в монолитной системе является типичной частью ядра, но в MINIX 3 может быть реализована как отдельный сервер из пространства пользователя.

Структура семафорного сервера в MINIX 3 показана на рис. 2. После инициализации сервер входит в основной цикл, который продолжается бесконечно. В каждой итерации сервер блокирует и ожидает поступления сообщения-запроса (request message). Как только сообщение получено, сервер проверяет запрос. Если тип запроса известен (допустим), то для обработки запроса вызывается связанная с ним функция обработки и результат возвращается для того, чтобы отправитель не был заблокирован. Неверные типы запросов немедленно приводят к неверному ответу.

Как говорилось выше, обычные пользовательские процессы в MINIX 3 ограничены в части синхронной посылки сообщений. Запрос заблокирует отправителя до тех пор, пока не придёт ответ. Мы воспользуемся этим при построении семафорного сервера. Для операций UP сервер просто увеличивает значение семафора и немедленно посылает ответ для того, чтобы отправитель мог продолжить работу. Для операций DOWN наоборот, ответ будет задержан до тех пор, пока значение семафора будет уменьшаться, таким образом эффективно блокируя отправителя до тех пор, пока он не будет должным образом синхронизирован. Семафор имеет связанную с ним очередь процессов (FIFO), для отслеживания блокированных процессов-отправителей. После операции UP очередь проверяется для того, чтобы посмотреть, разблокирован ли ожидающий процесс.

```
void semaphore_server( ) {

  message m;
  int result;
  /* Initialize the semaphore server. */
  initialize( );
  /* Main loop of server. Get work and process it. */
  while(TRUE) {
    /* Block and wait until a request message arrives. */
    ipc_receive(&m);
    /* Caller is now blocked. Dispatch based on message type. */
    switch(m.m_type) {
    case UP: result = do_up(&m); break;
    case DOWN: result = do_down(&m); break;
    default: result = EINVAL;
    }
    /* Send the reply, unless the caller must be blocked. */
    if (result != EDONTREPLY) {
      m.m_type = result;
      ipc_reply(m.m_source, &m);
    }
  }
}
```

Рис. 2. Основной цикл сервера, использующий один семафор *S*

Все серверы и драйверы имеют аналогичный основной цикл. Функция initialize() *вызывается один раз перед входом в основной цикл, но она здесь не показана. Функции обработки* do_up() и do_down() *приведены на Рис. 3.*

Используя структуру семафорного сервера, мы должны организовать и взаимодействие с ним пользовательских процессов. Как только сервер запустится, он будет готов обслуживать запросы. В принципе, программист может сконструировать сообщения-запросы и послать их новому серверу, используя ipc_request(), но такие мелочи обычно скрыты в системных библиотеках наряду с остальными POISIX-образными функциями. Как это обычно принято, новые библиотечные вызовы sem_up() и sem_down() следовало бы добавить в libc для обработки этих вызовов. И хотя наш демонстрационный пример описывает очень упрощённый семафорный сервер, он легко может быть расширен и для обеспечения соответствия семафорным спецификациям POSIX, и для обработки многих семафоров и т.п.

Модульная структура MINIX 3 помогает ускорить разработку семафорного сервера несколькими способами. Во-первых, он может использоваться независимо от остальной части ОС, просто как обычное пользовательское приложение. Когда же он будет закончен, он может быть скомпилирован как автономное приложение и динамически запускаться, чтобы стать частью операционной системы. Нет необходимости строить новое ядро или перезагружать систему, что предотвращает простои системы и защищает других пользователей от неожиданных сбросов системы, отключения от Интернета, почтовых и FTP серверов и т.п. Когда сервер запустится, его привилегии ограничиваются в соответствии с принципом минимальных полномочий, так что тестирование и отладка нового семафорного сервера могут быть выполнены без какого бы то ни было влияния на остальную часть системы. Как только он готов, сценарий запуска может быть сконфигурирован для автоматической загрузки семафорного сервера во время инициализации операционной системы.

```
int do_down(message *m_ptr) {


  /* Resource available. Decrement semaphore and reply. */
  if (s > 0) {
    s = s – 1; /* take a resource */
    return(OK); /* let the caller continue */
  }
  /* Resource taken. Enqueue and block the caller. */
  enqueue(m_ptr->m_source); /* add process to queue */
  return(EDONTREPLY); /* do not reply in order to block the caller */
}

int do_up(message *m_ptr) {

message m; /* place to construct reply message */
  /* Add resource, and return OK to let caller continue. */
  s = s + 1; /* add a resource */
  /* Check if there are processes blocked on the semaphore. */
  if (queue_size() > 0) { /* are any processes blocked? */
    m.m_type = OK;
    m.m_source = dequeue(); /* remove process from queue */
    s = s – 1; /* process takes a resource */
    ipc_reply(m.m_source, m); /* reply to unblock the process */
  }
  return(OK); /* let the caller continue */
}
```

Рис. 3. Операторы up **и** down **семафорного сервера**

Функции enqueue(), dequeue() *и* queue_size() *выполняют управление очередью (list management) и не показаны.*

Заключение

MINIX 3 является новой, полностью модульной операционной системой, сконструированной для того, чтобы быть сверхнадёжной. Как и в случае других новшеств, наше стремление к надёжности налагает некоторые дополнительные ограничения на условия выполнения программ, но мультисерверная среда MINIX 3 делает жизнь системного программиста гораздо легче. Цикл разработки оказывается короче, вынужденный простой системы более не является неизбежным, программный интерфейс оказывается более POSIX-образным, а тестирование и отладка становятся легче. Производительность труда программиста, скорее всего, увеличивается, а качество кода может быть улучшено за счёт лучшей отслеживаемости. Системный администратор также вознаграждается, поскольку MINIX 3 улучшает конфигурируемость и удобство обслуживания операционной системы. И, наконец, мы проиллюстрировали управляемую сообщениями модель программирования в MINIX 3 на примере создания простого семафорного сервера и обсудили вопрос о том, в чём и как его разработка выигрывает от модульности MINIX 3. Интересующиеся читатели могут загрузить MINIX 3 (включая все исходные тексты) с http://www.minix3.org. Более 50'000 человек уже скачали её; попробуйте и вы.

ССЫЛКИ

[1] T.J. Ostrand and E.J. Weyuker, “The Distribution of Faults in a Large Industrial Software System,” *Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis,* ACM, 2002, pp. 55–64.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An Empirical Study of Operating System Errors,” *Proceedings of the 18th ACM Symposium on Operating System Principles,* 2001, pp. 73–88.

[3] M.M. Swift, M. Annamalai, B.N. Bershad, and H.M. Levy, “Recovering Device Drivers,” *Proceedings of the 6th Symposium on Operating System Design and Implementation*, 2004, pp. 1–15.

[4] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, “The Performance of µ-Kernel–Based Systems,” *Proceedings of the 16th Symposium on Operating System Principles*, 1997, pp. 66–77.

[5] J. Liedtke, “On µ-Kernel Construction,” *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995, pp. 237–250.

[6] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, “Mach: A New Kernel Foundation for UNIX Development,” *Proceedings of the USENIX 1986 Summer Conference*, 1986, pp. 93–112.

[7] D.R. Cheriton, “The V Kernel: A Software Base for Distributed Systems,” *IEEE Software,* vol. 1, no. 2, 1984, pp. 19–42.

[8] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier, “A New Look at Microkernel–Based UNIX Operating Systems: Lessons in Performance and Compatibility,” *Proceedings of the EurOpen Spring 1991 Conference*, 1991, pp. 13–32.

[9] S. Mullender, G. Van Rossum, A.S. Tanenbaum, R. Van Renesse, and H. Van Staveren, “Amoeba: A Distributed Operating System for the 1990s,” *IEEE Computer Magazine,* vol. 23, no. 5, 1990, pp. 44–54. [10] E.W. Dijkstra, “Goto Statement Considered Harmful,” *Communications of the ACM*, vol. 11, no. 3, 1968, pp. 147–148.

Йоррит Н. Эрдер (Jorrit N. Herder) в настоящее время получает степень доктора философии на отделении компьютерных систем Департамента компьютерных наук университета Врийе, Амстердам. Его научные интересы связаны с разработкой и применением безопасных и надёжных операционных систем. Эрдер получил степень магистра естественных наук в университета Врийе. Связаться с ним можно по адресу jnherder@cs.vu.nl.

Герберт Бос (Herbert Bos) является доцентом отделения компьютерных систем Департамента компьютерных наук университета Врийе, Амстердам. Его научные интересы включают современные средства организации сетей (advanced networking technology), операционные системы и компьютерная безопасность. Бос получил степень доктора философии в Кембриджском университете. Связаться с ним можно по адресу bos@cs.vu.nl.

Бен Грас (Ben Gras) имеет степень магистра в области компьютерных наук университета Врийе, Амстердам, а до этого работал системным администратором и программистом. В настоящее время он приглашён университетом Врийе в группу компьютерных систем в качестве программиста для работы над проектом MINIX 3. Связаться с ним можно по адресу bjgras@cs.vu.nl.

Филип Хомбург (Philip Homburg) получил степень доктора философии университета Врийе в области глобальных распределённых систем (wide-area distributed systems). До присоединения к этому проекту он занимался вопросами виртуальной памяти, построения сетей и X Windows в проекте Minix-vmd и работал над усовершенствованными файловыми системами (advanced file systems) в проекте Logical Disk. Связаться с ним можно по адресу philip@cs.vu.nl.

Эндрю С. Таненбаум (Andrew S. Tanenbaum) является профессором в области компьютерных наук (computer science) университета Врийе, Амстердам. Его научные интересы связаны с операционными системами и компьютерной безопасностью. Таненбаум получил степень бакалавра в Массачусетском Технологическом Институте и степень доктора философии в Калифорнийском Университете в Беркли. Он является членом Института инженеров по электротехнике и электронике (IEEE) и членом Ассоциации по вычислительной технике (ACM). Связаться с ним можно по адресу ast@cs.vu.nl.