Архитектура ОС

Что такое операционная система

Управление ресурсами компьютера

Одно из определений операционной системы (ОС) гласит: операционная система --- это средство управления ресурсами компьютера. Можно выделить три уровня разделения ресурсов, на каждом из которых распределение и использование ресурсов компьютера происходит по-своему.

Однозадачный режим

Начальный уровень, на котором не нужны никакие дополнительные действия для обеспечения доступности ресурсов компьютера --- когда у нас есть ровно одна программа, которая полностью распоряжается всеми ресурсами компьютера, пока она не закончит свою работу. Такая система называется однозадачной, и то, насколько разумно она распоряжается аппаратными ресурсами, может повлиять только на работу единственной запущенной программы. Если программа написана плохо, например, пожирает память, то в конце концов она просто перестаёт работать, потому что оперативной памяти ей не хватит. Хорошо написанная программа должна успешно распоряжаться процессором, всей оперативной памятью и всеми внешними (по отношению к процессору) устройствами.

Много ли нужно программного обеспечения, чтобы позволить этой задаче эффективно использовать память и устройства и нужно ли создавать какое-то отдельное программное обеспечение для поддержки этого процесса? Возможный ответ --- "нет", так как запущенная программа, как правило, содержит в себе все необходимое ПО. Типичный пример -- это старинная игровая консоль (SEGA, SNES), использующая картриджи. Каждый картридж --- отдельная программа, загружающаяся и работающая по принципу однозадачной системы.

Унификация доступа

Однако даже при однозадачной системе обычно неразумно всю задачу организации доступа к ресурсам возлагать на прикладную программу, поскольку это сильно затрудняет их разработку.

Как минимум, хотелось бы унифицировать способ использования различных аппаратных ресурсов, например внешних устройств. Для решения этой задачи необходима некая прослойка между программой, которая пользуется ресурсами компьютера, и самими ресурсами. Эта прослойка будет скрывать от программы подробности того, как именно этими ресурсами воспользоваться. Рассмотрим это на примере графической платы: мы поменяли видеокарту, вместе с ней поменяли прослойку, которая обеспечивает доступ по стандартному (и не зависящему от конкретной карты) протоколу к ресурсам этой видеокарты, и прикладные программы не "заметили" изменений, хотя в действительности по шине PCI передаются уже принципиально другие данные.

Таким образом, даже на уровне однозадачной системы встает вопрос унификации доступа к ресурсам. Идея в том, чтобы ПО не принимало во внимание особенности аппаратного обеспечения, чтобы за это отвечала указанная прослойка. Зависящая от конкретной аппаратуры часть этой прослойки называется драйвером.

Многозадачный режим. Разделение доступа

Как только выясняется, что ресурсами компьютера пользуются несколько различных программ (назовём этот режим многозадачным), сразу возникает вопрос -- к каким ресурсам какие программы имеют доступ? Простейший вариант: пусть программы выполняются по одной друг за другом -- сначала отработает первая, затем вторая и т.д. Даже в этом случае, если они работают друг за другом и не знают о существовании друг друга, возникает вопрос организации дисциплины доступа к таким внешним ресурсам, которые существуют в течение их работы, например, к накопителю данных.

Разумеется, если программы работают по очереди, нет необходимости защищать, скажем, оперативную память, но общее для всех хранилище данных на диске хорошо бы организовать таким образом, чтобы одна программа знала о том, где хранит свои данные другая программа, для того, чтобы избегать коллизий. Если это файловая система, как в Linux, неплохо, если бы программа работала с именами файлов, и любая операция с некоторым файлом должна быть заложена в программу сознательно.

Еще более очевидна необходимость разделения ресурсов, когда мы имеем многозадачную систему с параллелизмом или псевдопараллелизмом (что вообще-то почти одно и то же с точки зрения разделения ресурсов). Параллелизм возможен при наличии нескольких процессоров, одновременно выполняющих разные программы; псевдопараллелизм -- это ситуация, когда эти программы в действительности работают последовательно, но для пользователя создается впечатление, что они работают параллельно, и происходит это так: один квант времени работает одна программа, потом она приостанавливается, следующий квант времени работает другая программа, следующая, потом снова начинает работать уже первая программа, и все снова. Выполняющуюся (загруженную в память) программу называют процессом.

Псевдопараллелизм --- наиболее распространенный вариант, его еще называют вытесняющей многозадачностью. Программы не обязательно выполняются в строгой последовательности, возможно (как в случае, когда приложение ожидает завершения операции ввода-вывода) изменение порядка их работы. В целом задача планирования процессов достаточно сложна, например 2007-й год прошёл под флагом революции в области планировщиков процессов в UNIX-системах (как в Linux, так в BSD-подобных), этой теме был посвящен отдельный семинар.

Таким образом, в многозадачных системах все ресурсы компьютера --- процессорное время, оперативная память, внешние устройства -- должны быть разделены между всеми работающими программами. Например, в случае псевдопараллелизма нужно создать дисциплину распределения доступа к процессору (например, мы можем захотеть, чтобы "дележ" процессорного времени был "справедливым" --- каждой программе поровну). Другим примером будет разделение оперативной памяти: нужно определить дисциплину распределения памяти, а так же организовать доступ к ОП таким образом, чтобы одна программа не могла случайно испортить содержимое участка памяти другой программы, поскольку эти программы не обязаны знать о том, что они сосуществуют вместе на одном компьютере. Третьим примером будет разделение принтера или сканера --- пока принтер полностью не завершил обработку одной заявки на печать, он не может начать обработку следующей.

Таким образом, все ресурсы, которые используют прикладные программы, должны быть явно обозначены, и должны существовать договоренности об их использовании, чтобы ситуация, когда одна программа модифицирует какой-то ресурс, принадлежащий другой программе, была исключительно штатной и контролируемой. Совокупность этих правил называют механизмом разделения ресурсов.

Разделение доступа к ресурсам

Как следует из проблем создания систем с параллельным выполнением программ, задача управления ресурсами оказывается связанной с задачей разделения доступа к ресурсам. Поэтому второе определения операционной системы гласит: операционная система --- это средство разделения ресурсов компьютера между прикладными программами.

Разделение центрального процессора

В персональных машинах чаще всего используется всего лишь один центральный процессор. Ресурсы ЦП включают в себя не только время его работы, но и объекты, связанные с хранением данных (регистры процессора). Это значит, что когда мы организуем псевдопараллелизм, мы должны предусмотреть не только ситуацию разбиения по времени (некоторое время работает одна, а некоторое --- другая задача), но и механизм сохранения тех элементов памяти, которые не дублируются для каждой программы, от одной программы, на то время, пока работает другая программа. К таким элементам относятся прежде всего регистры центрального процессора.

Всё состояние системы (в первую очередь --- центрального процессора), которое необходимо сохранить, чтобы в будущем продолжить выполнение процесса, называется контекстом процесса.

Это что значит, что в случае псевдо-параллелизма мы имеем один процессор, набор оперативной памяти, а в очереди сидят процесса (процесс-1, процесс-2 и так далее). С каждым созданным процессом связан контекст процесса, сохранённый в той же оперативной памяти. Когда процесс дожидается своей длои процессорного времени, то перед его выполнением, контекст подгружается, в частности, восстанавливаются значения регистров процессора. Затем процесс некоторое время (обычно порядка милисекунд) выполняется, после чего останавливается и контекст на момент прекращения выполнения сохраняется обратно. Затем подгружается контекст следующего по очереди процесса, он выполняется, и так далее. Даже такая с виду простая штука, как разделение ресурса под названием "центральный процессор" уже приводит к довольно сложным механизмам разделения доступа к этому ресурсу: квантование выполнения и сохранение контекста.

Разделение оперативной памяти

Тут ситуация чуть более простая, если в ней не разбираться, и чуть более сложная, если в ней разбираться досконально. #Никому не надо рассказывать слишком подробно, что На некотором уровне абстракции оперативная память представляет собой последовательность ячеек с номерами от 0 до самого большого значения, и сколько у нас воткнуто в плату памяти, столько и будет. Однако, в действительности всё значительно сложнее.

Существует несколько видов адресов памяти. В системах, которые ведут свое происхождение от однозадачных систем и не обеспечивают надлежащее разделение памяти между задачами (типичная такая система -- DOS, а также, с оговорками, Windows 3.*), основной являются физические адреса памяти --- одна программа занимает адреса от 1000 до 100000, другая от 120000 до 202000 байт, и так далее.

Если при этом одной программе по причине ошибки приходило в голову исписать нулями не свою, а "чужую" память (начав, к пример, в своей, а в результате ошибки убежав за границу), то тогда она убивала всю оперативную память, доступную операционной системе, и компьютер "зависал". Типичная ситуация в DOS: написал программу на языке Си с ошибкой, запустил, в итоге перегрузил компьютер. В младших версиях Windows наблюдалось то же самое, причём отдельные рецидивы были вплоть до Windows 98 (но не в линейке полноценных ОС Windows NT).

Операционные системы, которые обеспечивают нормальное разделение памяти, пользуются аппаратным страничным механизмом разделения памяти, котому уже около 30-ти лет, а в недорогих компьютерах он появился вместе с процессором Intel i386 почти четверть века назад. Устроен этот механизм следующим образом. Вся оперативная память делится на страницы некоторого фиксированного размера (обычно по 4096 байт). Пусть запускаются два процесса в псевдо-параллельном режиме, причём никто из них не ведёт обмен с внешними устройствами, а жаждет лишь выполняться на процессоре. Для каждого создается контекст процесса, набор регистров, который нужно хранить, и все остальное, потом каждый из них запрашивает себе память, которая ему выделяется. Куски памяти, запрошенные разными процессами, с физической точки зрения перемешаны совершенно произвольно. Но реальных адресов с 0 до полного объема памяти ни один процесс не наблюдает. У него происходит аппаратно поддерживаемое со стороны процессора преобразования виртуального адреса, с которым они работают, в адрес физический.

При этом все страницы памяти, принадлежащие процессу, складываются в некоторую т.н. виртуальную память с непрерывной адресацией (допустим, объёмом 100 Mb), которую, собственно, и "видит" этот процесс. В таких случаях в понятие контекста процесса включается указанное преобразование, поскольку для каждого процесса оно своё. Когда процесс обращается к памяти, он видит только ту память, которую ему выдали, от нулевого адреса и до конца запрошенной памяти (видит он даже до физически максимального адреса). Доступ же в память напрямую, по произвольному физическому адресу, процессу запрещён.

Из скольких страниц состоит выделенная процессу память и какие у них физические адреса --- процессор не волнует, это проблема операционной системы. Пусть у нас всего 1 Гб памяти, процесс заказал 100 Мб, он 100 Мб получил, а эти 100 Мб собраны из 25600-ти страниц памяти по 4 Кб каждая.

Аналогичная ситуация происходит со вторым процессом программой, который так же видит только свою память, строго от нуля, чужую память он не видит в принципе. Таким образом, при использовании виртуальной памяти программа не может случайно испортить чужую память. По предварительной договорённости можно обратиться в вышестоящей организации и сказать: у меня есть две программы, которые используют общий сегмент памяти, разреши им пользоваться, потому что так проще данные передавать: одна запишет, другая возьмёт. Операционная ситсема ответит: хорошо, тогда вот вам еще один сегмент, который будет подключен в адресное пространство обоих процессов (со своими адресами). Когда один процесс в него пишет, другой процесс сразу видит изменения. Но, в отличие от ситуации, когда программа расписала нулями всю чужую память, эта ситуация --- контролируемая. Вы написал программы так, что они заказали себе один и тот же сегмент, и по Вашей воле производят там изменения.

Ещё один достоинством страничной организации памяти и виртальных адресов является возможность записать малоиспользыемые процессом таблицы памяти на диск, а при обращении к ним считать их обратно в память, причём все это совершенно незаметно для процесса. Так устроена подкачка памяти с диска ("своп").

Разделение внешних устройств

Если система многозадачная, то разделения требуют вообще все ресурсы, не только процессор как время и как недублируемая память, и не только оперативная память как объект желаний процессов. Разделения требует, например, звуковая карта, т.к. если несколько программ пытаются играть звук, это, во-первых, отвратительно слышно, во-вторых, если не организован процесс разделения, то первая программа может получить доступ к ресурсу, а все остальные могут его и не получить. Если операционная система предоставляет некоторый интерфейс к звуковой карте, это значит, что либо в каждый момент времени по отношению к звуковой карте она "однозадачная": первая по порядку запуска программа

получает полный доступ, а всем остальным не хватило, т.е. имеет место монопольный доступ, либо она как-то решает проблему разделения звуковой карты. Для решения этой проблемы система может поддерживать несколько виртуальных звуковых карт, а каждой процесс будет работать со своей виртуальной картой. Затем система организует наложение друг на друга звуков от нескольких процессов и передачу результатов физической звуковой карте. То же самое касается видеокарты, периферийных устройств и так далее, причём в случае принтера "наложение" недопустимо, а в случае видеокарты происходит сплошь и рядом.

Ограничение доступа

Задачи операционной системы не ограничиваются обеспечением доступа к ресурсом и их разделением между программами. В случае многозадачной системы постоянно возникает ограничить ряд процессов таким образом, чтобы они не смогли получить доступ к каким-либо ресурсам в принципе. Пример: допустим я храню пароли в текстовом файле и не хочу, чтобы их мог подсмотреть процесс, запущенный любым другим пользователем и вообще любой процесс, которому я не доверяю.

В результате решения этой проблемы мы приходим к тому, что система не просто многозадачная, но и задачи бывают разные с точки зрения уровня доступа. Про одни задачи известно, что им можно иметь доступ к некоторому ресурсу, а про другие известно, что нельзя. В простейшем случае задачи делятся на группы, принадлежащие разным пользователям, причём это могут быть как фиктивные системные пользователи, так и пользователи, связанные с пользователями-людьми. Одной группе задач пользователь разрешает иметь доступ к некоторому ресурсу, а другим группам --- не разрешает. И мы переходим от системы многозадачной к системе многозадачной и многопользовательской.

Если в случае многозадачной системы все задачи была с равными правами и главной целью системы было обеспечить невозможность случайного доступа к чужим ресурсам, то в случае многопользовательской системы разные задачи могут иметь разные права, и основной целью будет недопущение не просто случайного, а несанкционированного доступа к ресурсу. Если один пользователь не хочет, чтобы задачи другого пользователя имели доступ к ресурсам первого пользователя, то он это должен уметь запретить. Это задача называется ограничение доступа к ресурсам.

Что такое операционная система

Мы получаем три группы задач, которые должна решать операционная система:

- унификация доступа;
- разделение доступа;
- ограничение доступа.

Фактически, это определение операционной системы: операционная система --- это программное обеспечение, которое обеспечивает выполнение трёх указанных задач для того аппаратного обеспечения, на котором эта ОС запущена. Не надо также забывать о том, что наша задача -- повысить удобство работы с компьютером. Ради чего были сделаны операционные системы? Ради того, чтобы человек, который создаёт прикладные программы или их запускает не занимался решением указанных задач самостоятельно и вручную.

Вообще говоря, нет полностью удовлетворительного определения ОС. Более того, если уйти в историю, мы поймём, что пользуемся этим словом неправильно. Operating system -- это не ПО, это систематическое (system) руководство по использованию (operating) компьютера, совокупность методов. Как правило, эти методы программируются и получается программное обеспечение, но смысл термина изначально был именно такой.

Примеры

Из приведённого определения выкинуты подробности отдельных существующих представителей операционных систем. Хотелось бы избежать ситуации, когда под операционной системой понимается нечто с кнопкой "Пуск" в нижнем левом углу экрана. Во-первых, никакого экрана может вообще не быть, поскольку нет графических ресурсов у данной аппаратной платформы (ближайший пример --- домашний ADSL-модем или маршрутизатор). Во-вторых, может быть экран есть, но нету графической кнопки для нажатия на нее мышкой, т.к. может случиться, что эта аппаратная платформа управляется без помощи графических инструментов, а исключительно подачей ей команд, а всякая графика -- прикладная, а не интерфейсная (пример: интерфейс управления самолётом).

Кроме того, термин "ОС" применялся и к сущностям, которые турдно отнести к операционным системам. Приведём пример DOS, всё ещё знакомый многим. Почему DOS -- это не ОС? DOS -- это, дословно, дисковая операционная система, если кто помнит. Многозадачная (в памяти могло быть несколько программ, но без квантования времени и вытеснения задач), а так же однопользовательская. DOS не умеет обеспечивать защиту памяти, со стороны ОС не предоставляются возможности обеспечить даже защиту от случайного, непреднамеренного доступа. Не предоставляет нормальных способов разделения таких ресурсов как видео, аудио, или даже последовательный порт. Для решения этих задача каждый разработчик должен был написать специальную программу-драйвер или, что еще хуже, вписывать в свою программу работу непосредственно с регистрами видеокарты, только после чего всё и начинает работать.

Единственные инструменты по унификации и разделению ресурсов, которые предоставляет DOS пользователю, это файловая система, но она и в те времена вызывала уже много нареканий. Таким образом, DOS содержит только зачатки функций операционной системы. В настоящее время существуют специализированные операционные системы, которые, несмотря на своё название, не выполняют всех поставленных нами задач, поскольку являются компромиссным вариантом и пытаются достигнуть, например, наименьшего времени отклика системы. Все современные ОС общего назначения, в том числе линейка Windows NT, GNU/Linux и BSD-системы вместе с Mac OS X на все поставленные здесь вопросы отвечают полностью утвердительно.

Пользователи системы

Между включением питания компьютера и моментом, когда система готова к работе с пользователем, происходит процедура **загрузки системы**. В процессе загрузки будет запущена основная управляющая программа (**ядро**), определено и инициализировано имеющееся оборудование, активизированы сетевые соединения, запущены системные службы. В Linux во время загрузки на экран выводятся диагностические сообщения о происходящих событиях, и если всё в порядке и не возникло никаких ошибок, загрузка завершится выводом на экран приглашения `"login:"`. Оно может быть оформлено по-разному, в зависимости от настройки системы оно может отображаться в красиво оформленном окне или в виде простой текстовой строки вверху экрана. Это приглашение к **регистрации в системе**: система ожидает, что в ответ на это приглашение будет введено **входное имя пользователя**, который начинает работу. Естественно, имеет смысл вводить такое имя, которое уже известно системе, чтобы она могла "узнать", с кем предстоит работать, выполнять команды "незнакомого" Linux откажется.

Многопользовательская модель разграничения доступа

Процедура регистрации в системе *обязательна* для Linux, работать в системе, не зарегистрировавшись под тем или иным именем пользователя, просто *невозможно*(1). Для каждого пользователя определена сфера его полномочий в системе: программы, которые он может запускать, файлы, которые он имеет право просматривать, изменять, удалять. При попытке сделать что-то, выходящее за рамки полномочий, пользователь получит сообщение об ошибке. Такая строгость может показаться необязательной, если пользователи компьютера доверяют друг другу, и особенно если у компьютера только один пользователь. Такая ситуация очень распространена на сегодняшний день, когда слово "компьютер" означает в первую очередь "персональный компьютер".

Однако **персональный компьютер** -- довольно-таки позднее явление в мире вычислительной техники, получившее широкое распространение только в последние два десятилетия. Несколько раньше "компьютер" ассоциировался с огромным и дорогостоящим (занимавшем целые залы) вычислительным центром, предназначенным в первую очередь для решения разного рода научных задач. Машинное время такого центра стоит очень недёшево, и при этом его возможности необходимы одновременно многим сотрудникам, которые могут ничего не знать о работе друг друга. Требуется следить за тем, чтобы не произошло случайного вмешательства пользователей в чужую работу и повреждения чужих данных (файлов), выделять каждому машинное время (по возможности избежав простаивания), пространство на диске и при этом не допустить узурпирования всех ресурсов одним пользователем и его задачей, а равномерно делить ресурсы между всеми. Для такой системы принципиально важно знать, кому принадлежат задачи и файлы, поэтому и возникла необходимость выдавать доступ к ресурсам системы только после того, как пользователь *зарегистрируется в системе* под тем или иным именем.

Такая модель была реализована в **многопользовательской операционной системе** UNIX. Именно от неё Linux -- также многопользовательская система -- унаследовал принципы работы с пользователями. Но это не просто дань традиции или стремление к универсальности: многопользовательская модель позволяет решить ряд задач, весьма актуальных и для современных **персональных компьютеров**, и для серверов, работающих в локальных и глобальных сетях, и вообще в любых системах, одновременно выполняющих *разные* задачи, отвечают за которые *разные* люди.

Компьютер -- это всего лишь инструмент для решения разного рода прикладных задач: от набора и распечатывания текста до вычислений. Сложность состоит в том, что для изменения этого инструмента и для работы с его помощью используются одни и те же операции: изменение файлов, выполнение программ. Получается, что, если не соблюдать осторожности, побочным результатом работы может стать выход из строя самой системы. Поэтому первоочередная задача для систем любого масштаба -- разделять повседневную работу и изменение самой системы. В многопользовательской модели эта задача решается очень просто: разделяются **обычные пользователи** и **администратор (ы)**. В полномочия обычного пользователя входит все необходимое для выполнения прикладных задач, попросту говоря, для работы, однако ему запрещено выполнение действий, изменяющих саму систему. Таким образом можно избежать повреждения системы в результате ошибки пользователя (нажал не ту кнопку) или ошибки в программе, или даже по злому умыслу (например, вредительской программой-вирусом). Полномочия администратора обычно не ограничены.

Для персонального компьютера, с которым работают несколько человек, довольно важно обеспечить каждому независимую рабочую среду. Это снижает вероятность случайного повреждения чужих данных, а также позволяет каждому пользователю настроить внешний вид рабочей среды по своему вкусу и, например, сохранить расположение открытых окон между сеансами работы. Эта задача очевидным образом решается в многопользовательской модели: организуется **домашний каталог**, где хранятся данные пользователя, настройки

внешнего вида и поведения его системы и т. п., доступ остальных пользователей к этому каталогу ограничивается.

Если компьютер подключён к глобальной или локальной сети, то вполне вероятно, что какую-то часть хранящихся на нем ресурсов имеет смысл сделать публичной и доступной по сети. Напротив, часть данных, скорее всего, делать публичными не следует (например, личную переписку). Ограничив публичный доступ пользователей к персональным данным друг друга, мы решим и эту задачу.

Именно благодаря гибкости многопользовательской модели разграничения доступа она используется сегодня не только на серверах, но и на домашних персональных компьютерах. В самом простом варианте -- для персонального компьютера, на котором работает только один человек -- эта модель сводится к двум пользователям: обычному пользователю для повседневной работы и администратору -- для настройки, обновления, дополнения системы и исправления неполадок. Но даже в таком сокращённом варианте это даёт целый ряд названных выше преимуществ.

Учётные записи

Конечно, система может быть "знакома" с человеком только в переносном смысле: в ней должна храниться запись о пользователе с таким именем и о связанной с ним системной информации -- **учётная запись**. Английский эквивалент термина **учётная запись** -- **account**, "счёт". Именно с учётными записями, а не с самими пользователями, и работает система. В действительности, соотношение учётных записей и пользователей-людей в Linux обычно не является однозначным: несколько человек могут использовать одну учётную запись -- система не может их различить. И в то же время в Linux имеются учётные записи для **системных пользователей**, от имени которых работают некоторые программы и которые не предназначены для работы людей.

учётная запись — Объект системы, при помощи которого Linux ведёт учёт работы пользователя в системе. Учётная запись содержит данные о пользователе, необходимые для регистрации в системе и дальнейшей работы с ней.

Учётные записи могут быть созданы во время установки системы или после установки. Подробно процедура создания учётных записей (добавления пользователей) описана в лекции Конфигурационные файлы.

Главное для человека в учётной записи -- её название, **входное имя пользователя**. Именно о нём спрашивает система, выводя приглашение "`login:`". Помимо входного имени в учётной записи содержатся некоторые сведения о пользователе, необходимые системе для работы с ним. Ниже приведён список этих сведений.

входное имя — Название учётной записи пользователя, которое нужно вводить при регистрации пользователя в системе.

Идентификатор пользователя

Linux связывает **входное имя** с **идентификатором пользователя** в системе -- **UID** (User ID). **UID** -- это положительное целое число, по которому система и отслеживает пользователей(2). Обычно это число выбирается автоматически при регистрации учётной записи, однако оно не может быть совершенно произвольным. В Linux есть некоторые соглашения относительно того, каким типам пользователей могут быть выданы идентификаторы из того или иного диапазона. В частности, **UID** от "`0`" до "`100`" зарезервированы для **псевдопользователей**(3).

идентификатор пользователя	Уникальное число, однозначно идентифицирующее **учётную запись** пользователя в Linux. Таким числом снабжены все **процессы** Linux и все объекты **файловой системы**. Используется для персонального учёта действий пользователя и определения **прав доступа** к другим объектам системы

Идентификатор группы

Кроме идентификационного номера пользователя с учётной записью связан **идентификатор группы**. **Группы пользователей** применяются для организации доступа нескольких пользователей к некоторым ресурсам. У группы, так же, как и у пользователя, есть имя и идентификационный номер -- **GID** (Group ID). В Linux пользователь должен принадлежать как минимум к одной группе -- **группе по умолчанию**. При создании учётной записи пользователя обычно создаётся и группа, имя которой совпадает с **входным именем**(4), именно эта группа будет использоваться как группа по умолчанию для этого пользователя. Пользователь может входить более чем в одну группу, но в учётной записи указывается только номер группы по умолчанию.

Полное имя

Помимо **входного имени** в учётной записи содержится и **полное имя** (имя и фамилия) использующего данную учётную запись человека. Конечно, пользователь может указать что угодно в качестве своего имени и фамилии. Полное имя необходимо не столько системе, сколько людям -- чтобы иметь возможность определить, кому принадлежит учётная запись.

Домашний каталог

Файлы всех пользователей в Linux хранятся раздельно, у каждого пользователя есть собственный **домашний каталог**, в котором он может хранить свои данные. Доступ других пользователей к домашнему каталогу пользователя может быть ограничен. Информация о домашнем каталоге обязательно должна присутствовать в учётной записи, потому что именно с него начинает работу пользователь, зарегистрировавшийся в системе.

Командная оболочка

Каждому пользователю нужно предоставить способ взаимодействовать с системой: передавать ей команды и получать её ответы. Для этой цели служит специальная программа -- **командная оболочка** (или **интерпретатор командной строки**), она должна быть запущена для каждого пользователя, зарегистрировавшегося в системе. Поскольку в Linux доступно несколько разных командных оболочек, в учётной записи указано, какую из командных оболочек нужно запустить для данного пользователя. Если специально не указывать командную оболочку при создании учётной записи, она будет назначена по умолчанию, вероятнее всего это будет `bash`.

интерпретатор командной строки	Программа, используемая в Linux для организации диалога человека и системы. Командный интерпретатор имеет три основных ипостаси: (1) редактор и анализатор команд в **командной строке**, (2) высокоуровневый системно-ориентированный язык программирования, (3) средство организации взаимодействия команд друг с другом и с системой.

Понятие "администратор"

В Linux есть ровно один пользователь, полномочия которого в системе принципиально отличаются от полномочий остальных пользователей -- это пользователь с идентификатором "`0`". Обычно учётная запись пользователя с UID=0 называется `root` (англ., "корень"). Пользователь `root` -- это "администратор" системы Linux, учётная запись для `root` обязательно присутствует в любой системе Linux, даже если в ней нет никаких других учётных записей. Пользователю с таким UID разрешено выполнять *любые* действия в системе, а значит, любая ошибка или неправильное действие может повредить систему, уничтожить данные и привести к другим печальным последствиям. Поэтому *категорически* не рекомендуется регистрироваться в системе под именем `root` для повседневной работы. Работать в `root` следует только тогда, когда это действительно необходимо: при настройке и обновлении системы, восстановлении после сбоев.

Именно `root` обладает достаточными полномочиями для создания новых учётных записей.

Регистрация в системе

Вернёмся теперь к нашей загруженной операционной системе Linux, которая по-прежнему ожидает ответа на своё приглашение "`login:`". Если Ваша система настроена таким образом, что это приглашение оформлено в виде графического окна в центре экрана, нажмите комбинацию клавиш `Ctrl+Alt+F1` -- произойдёт переключение видеорежима и Вы увидите перед собой чёрный экран с примерно следующим текстом:

```
Welcome to Some Linux / tty1

localhost login:
```

Начальное приглашение к регистрации

Мы переключились в так называемый **текстовый режим**, в котором нам недоступны возможности графических интерфейсов: рисование окон произвольной формы и размера, поддержка миллионов цветов, отрисовка изображений. Все возможности текстового режима ограничены набором текстовых и псевдографических символов и несколькими десятками базовых цветов. Однако в Linux в текстовом режиме можно выполнять практически любые действия в системе (кроме тех, которые требуют непосредственного *просмотра* изображений). Текстовый режим в Linux -- это полнофункциональный способ управления системой. В различных реализациях Linux работа в графическом режиме может выглядеть очень по-разному(5), более того, графический режим может быть даже недоступен после установки системы без специальной настройки. Текстовый режим, напротив, доступен в любой реализации Linux и всегда выглядит практически одинаково. Именно поэтому все дальнейшие примеры и упражнения мы будем проделывать в текстовом режиме, возможностей которого будет достаточно для освоения всего излагаемого в курсе материала.

Первая строка в примере -- это просто приглашение, она носит информационный характер. Существует очень много различных реализаций Linux (существующие реализации будут обсуждаться в лекции Политика свободного лицензирования. История Linux: от ядра к дистрибутивам), и в каждом из них принят свой формат первой пригласительной строки, хотя почти наверняка там будет указано, с какой именно версией Linux Вы имеете дело, и, возможно, будут присутствовать ещё некоторые параметры. В наших примерах мы будем использовать условную реализацию Linux -- "Some Linux".

Вторая строка начинается с **имени хоста** -- собственного имени системы, установленной на данном компьютере. Это имя существенно в том случае, если компьютер подключён к

локальной или глобальной сети, если же он ни с кем более не связан, это имя может быть любым. Обычно имя хоста определяется уже при установке системы, однако в нашем случае этого сделано не было, и используется вариант по умолчанию -- "`localhost`". Заканчивается эта строка собственно приглашением к регистрации в системе -- словом "`login:`".

Теперь понятно, что в ответ на это приглашение мы должны ввести **входное имя**, для которого есть соответствующая **учётная запись** в системе. В правильно установленной операционной системе Linux должна существовать как минимум одна учётная запись для **обычного пользователя**. Во всех дальнейших примерах у нас будет участвовать Мефодий Кашин, владелец учётной записи "`methody`" в системе "Some Linux". Вы можете пользоваться для выполнения примеров любой учётной записью, которая создана в Вашей системе (естественно, кроме `root`).

Итак, Мефодий вводит своё входное имя в ответ на приглашение системы:

```
Welcome to Some Linux / tty1
localhost login: Methody

Password:
Login incorrect

login:
```

Регистрация в системе

В ответ на это система запрашивает **пароль**. Пароль Мефодия нам неизвестен, поскольку он его никому не говорит. Когда Мефодий вводил свой пароль, на экране монитора он не отображался (это сделано, чтобы пароль нельзя было подсмотреть), однако Мефодий точно знает, что не сделал опечатки. Тем не менее система отказала ему в регистрации, выдав сообщение об ошибке ("`Login incorrect`"). Если же внимательно посмотреть на введённое им имя пользователя, можно заметить, что оно начинается с заглавной буквы, в то время как учётная запись называется "`methody`". Linux всегда делает различие между заглавными и строчными буквами, поэтому "`Methody`" для него -- уже другое имя. Теперь Мефодий повторит попытку:

```
login: methody
Password:
[methody@localhost methody]$
```

Успешная регистрация в системе

В этот раз регистрация прошла успешно, о чём свидетельствует последняя строка примера -- **приглашение командной строки**. Приглашение -- это подсказка, выводимая **командной оболочкой** и свидетельствующая о том, что система готова принимать команды пользователя. Приглашение может быть оформлено по-разному, более того, пользователь может сам управлять видом приглашения (подробнее это будет рассмотрено в лекции Возможности командной оболочки), но почти наверняка в приглашении содержатся **входное имя** и **имя хоста** -- в нашем примере это "`methody`" и "`localhost`" соответственно. Заканчивается приглашение чаще всего символом "`$`". Это **командная строка**, в которой будут отображаться все введённые пользователем с клавиатуры команды, а при нажатии на клавишу Enter содержимое командной строки будет передано для исполнения системе.

Идентификация (authentication)

Когда система выводит на экран приглашение командной строки после того, как правильно введены имя пользователя и пароль, это означает, что произошла **идентификация пользователя** (authentication, "проверка подлинности"). Пароль может показаться излишней сложностью, но у системы нет другого способа удостовериться, что за монитором находится именно тот человек, который имеет право на использование данной учётной записи.

Конечно, процедура идентификации имеет очевидное значение для систем, к которым имеют непосредственный или сетевой доступ многие не связанные друг с другом пользователи. Процедура идентификации даёт уверенность, что к такой системе не получит доступ случайный человек, не имеющий права использовать её ресурсы и хранящуюся там информацию. Одновременно она даёт определённую гарантию безопасности от злонамеренного вмешательства: даже если навредить попытается пользователь, имеющий учётную запись, его действия будут зарегистрированы в системе (поскольку системе всегда известно, от имени какой учётной записи выполняются те или иные действия), и злоумышленника можно будет найти и остановить.

Для тех пользователей, кому процедура идентификации кажется утомительной и необязательной (например, единственным пользователям персональных компьютеров), существует возможность получить доступ к системе, минуя процедуру идентификации. Для этой цели служит программа `autologin`. Она предоставляет доступ к работе с графическим интерфейсом сразу после загрузки системы, не запрашивая имя пользователя и пароль. В действительности, `autologin` запускает все программы от имени одного пользователя, зарегистрированного в системе. Например, Мефодий мог бы использовать свою учётную запись `methody` для автоматического входа в систему. Однако у этого подхода есть свои минусы:

- Теряется возможность определить, кто, что и когда делал в системе, потому что все реальные пользователи работают с одной учётной записью, с точки зрения системы все они -- один и тот же пользователь.
- Вся личная информация этого пользователя становится "общественной".
- Пароль легко забывается (пароль всё равно есть у любого пользователя), потому что его не нужно вводить каждый день. При этом `autologin` даёт доступ только человеку, сидящему перед монитором и только к работе с графическим интерфейсом. Если же потребуется, например, скопировать файлы с Вашего компьютера по сети, пароль всё равно нужно будет вводить.

Учитывая все перечисленные минусы, можно заключить, что использовать `autologin` разумно только в тех системах, которые не подключены к локальной или глобальной сети, и к которым при этом открыт публичный доступ (например, в библиотеке).

Смена пароля

Если учётная запись была создана не самим пользователем, а администратором многопользовательской системы (скажем, администратором компьютерного класса), скорее всего был выбран тривиальный пароль с тем расчётом, что пользователь его изменит при первом же входе в систему. Распространены тривиальные пароли "`123456`", "`empty`" и т. п. Поскольку пароль -- это единственная гарантия, что Вашей учётной записью не воспользуется никто, кроме Вас, есть смысл выбирать в качестве пароля неочевидные последовательности символов. В Linux нет серьёзных ограничений на длину пароля или входящие в него символы (в частности, использовать пробел *можно*), но нет смысла делать пароль слишком длинным -- сразу возрастает опасность его забыть. Надёжность паролю придаёт его *непредсказуемость*, а не длина. Например, пароль, представляющий собой Ваше

имя или повторяющий название учётной записи, *очень предсказуем*. Наименее предсказуемы пароли, представляющие собой случайную комбинацию прописных и строчных букв, цифр, знаков препинания, но их и труднее всего запомнить.

Пользователь может в любой момент поменять свой пароль. Единственное, что требуется для смены пароля -- знать текущий пароль. Допустим, Мефодий придумал более удачный пароль и решил его поменять. Он уже зарегистрирован в системе, поэтому ему нужно только набрать в командной строке команду `passwd` и нажать Enter.

```
[methody@localhost methody]$ passwd
Changing password for methody.
Enter current password:

You can now choose the new password or passphrase.

A valid password should be a mix of upper and lower case letters,
digits, and other characters.  You can use an 8 character long
password with characters from at least 3 of these 4 classes, or
a 7 character long password containing characters from all the
classes.  An upper case letter that begins the password and a
digit that ends it do not count towards the number of character
classes used.

A passphrase should be of at least 3 words, 12 to 40 characters
long and contain enough different characters.

Alternatively, if noone else can see your terminal now, you can
pick this as your password: "spinal&state:buy".

Enter new password:
```

Смена пароля

Набрав в командной строке "`passwd`", Мефодий запустил программу `passwd`, которая предназначена именно для замены информации о пароле в учётной записи пользователя. Она вывела приглашение ввести текущий пароль ("`Enter current password`"), а затем, в ответ на правильно введённый пароль, предложила подсказку про грамотное составление пароля и даже вариант надёжного пароля, который Мефодий вполне может использовать, если никто в данный момент не видит его монитора. При каждом запуске `passwd` генерирует новый случайный пароль и предлагает его пользователю. Однако Мефодий не воспользовался подсказкой и придумал пароль сам.

```
Enter new password:
Weak password: not enough different characters or classes for this length.
Try again.

. . .

Enter new password:
```

Смена пароля (продолжение)

В данном случае операция не удалась, поскольку с точки зрения `passwd` пароль, придуманный Мефодием, оказался слишком простым(6). В следующий раз ему придётся ввести более сложный пароль. `passwd` запрашивает новый пароль дважды, чтобы удостовериться, что в первый раз не было сделано опечатки, если же всё в порядке, она выведет сообщение о том, что операция смены пароля прошла успешно, и завершит работу,

вернув Мефодию приглашение командной строки.

```
Enter new password:
Re-type new password:
passwd: All authentication tokens updated successfully
[methody@localhost methody]$
```

Пароль изменён

Придирчивость, с которой `passwd` относится к паролю пользователя, не случайна. Пароль пользователя -- одно из самых важных и зачастую одно из самых слабых мест безопасности системы. Отгадавший Ваш пароль (причём не имеет значение, человек это сделал или злонамеренная программа) получит доступ к ресурсам системы ровно в том объёме, в котором он предоставляется Вам, сможет читать и удалять Ваши файлы и т. п. Особенно это важно в случае пароля администратора, потому что его полномочия в системе гораздо шире, а действия от его имени могут повредить и саму систему. Обычному пользователю в некоторых обстоятельствах также могут быть переданы полномочия администратора (этот вопрос будет подробно обсуждаться в лекции Права доступа), в таком случае не менее важно, чтобы и его пароль был надёжным.

Пароль пользователя `root` изначально назначается при установке системы, однако он может быть изменён в любой момент впоследствии точно так же, как и пароль обычного пользователя.

Одновременный доступ к системе

То, что Linux -- многопользовательская и многозадачная система, проявляется не только в разграничении прав доступа, но и в организации рабочего места. Каждый компьютер, на котором работает Linux, предоставляет возможность зарегистрироваться и получить доступ к системе одновременно нескольким пользователям. Даже если в распоряжении всех пользователей есть только один монитор и одна системная клавиатура, эта возможность небесполезна: одновременная регистрация в системе нескольких пользователей позволяет работать по очереди без необходимости каждый раз завершать все начатые задачи (закрывать все окна, прерывать исполнение всех программ) и затем возобновлять их. Более того, ничто не препятствует зарегистрироваться в системе несколько раз под одним и тем же **входным именем**. Таким образом, можно получить доступ к одним и тем же ресурсам (своим файлам) и организовать параллельную работу над несколькими задачами.

Виртуальные консоли

Характерный для Linux способ организации параллельной работы пользователей -- **виртуальные консоли**.

Допустим, что Мефодий хочет зарегистрироваться в системе ещё раз, чтобы иметь возможность следить за выполнением двух программ одновременно. Он может сделать это, не покидая текстового режима: достаточно нажать комбинацию клавиш `Alt+F2`, и на экране появится новое приглашение к регистрации в системе.

```
Welcome to Some Linux / tty2
localhost login: methody
Password:
[methody@localhost methody]$
```

Вторая виртуальная консоль

Мефодий ввёл свой новый пароль и получил приглашение командной строки, аналогичное тому, которое мы уже видели в предыдущих примерах. Нажав комбинацию клавиш `Alt+F1`, Мефодий вернётся к только что покинутой им командной строке, в которой он выполнял команду `passwd` для смены пароля. Приглашение в обоих случаях выглядит одинаково, и это не случайно -- обе командные строки предоставляют совершенно эквивалентный доступ к системе, в любой из них можно выполнять любые доступные команды.

Наблюдательный Мефодий обратил внимание, что в последнем примере (tty2) первая строка приглашения оканчивается словом "`tty2`". "`tty2`" -- это обозначение *второй* **виртуальной консоли**. Можно переключаться между виртуальными консолями так, как если бы Вы переходили от одного монитора с клавиатурой к другому, подавая время от времени команды и следя за выполняющимися там программами. По умолчанию в Linux доступно не менее 6-ти виртуальных консолей, переключаться между которыми можно при помощи сочетания клавиши `Alt` с одной из функциональных клавиш (`F1`--`F6`), с каждым сочетанием связана соответствующая по номеру виртуальная консоль. Виртуальные консоли обозначаются "`ttyN`", где "`N`" -- номер виртуальной консоли.

виртуальная консоль	Виртуальные консоли -- это несколько параллельно выполняемых операционной системой программ, предоставляющих пользователю возможность зарегистрироваться в системе в текстовом режиме и получить доступ к командной строке.

Во многих дистрибутивах Linux одна из виртуальных консолей по умолчанию не может быть использована для регистрации пользователя, однако она не менее, если не более полезна. Если Мефодий нажмёт `Alt+F12`, он увидит консоль, заполненную множеством сообщений системы о происходящих событиях. В частности, там он может обнаружить две записи о том, что в системе зарегистрирован пользователь "`methody`". На эту консоль выводятся сообщения обо всех важных событиях в системе: регистрации пользователей, выполнении действий от имени администратора (`root`), подключении устройств и подгрузке драйверов к ним и многое другое.

Пример двенадцатой виртуальной консоли показывает, что виртуальные консоли -- довольно гибкий механизм, поддерживаемый Linux, при помощи которого можно решать разные задачи, а не только организацию одновременного доступа к системе. Для того, чтобы на виртуальной консоли появилось приглашение `login:` после загрузки системы, для каждой такой консоли должна быть запущена программа `getty`. Попробуйте нажать `Alt+F10` -- с большой вероятностью Вы увидите просто чёрный экран. Десятая виртуальная консоль поддерживается системой, однако чёрный экран означает, что для этой консоли не запущена никакая программа, поэтому воспользоваться её существованием не получится. Для каких именно консолей будет запущена программа `getty` -- определяется настройкой конкретной системы. Впоследствии эта настройка может быть изменена пользователем. О том, как это может быть сделано, речь пойдёт в лекции Этапы загрузки системы.

Графические консоли

Впрочем, как ни широки возможности текстового режима, Linux ими не ограничен. Подробно работа в графическом режиме будет разбираться в последующих лекциях (см. лекцию Графический интерфейс (X11)). Сейчас важно заметить, что если при загрузке системы приглашение "`login:`" было представлено в виде графического окна, можно вернуться к этому приглашению, нажав комбинацию клавиш `Ctrl+Alt+F7`. Процедура регистрации здесь будет совершенно аналогична регистрации в текстовом режиме. С той

разницей, что после **идентификации** пользователя (правильно введённого имени пользователя и пароля) Вы увидите не приглашение командной строки, а графическую рабочую среду. Как именно она будет выглядеть -- зависит от того, какую систему Вы используете, и как она настроена.

Кроме того, что несколько пользователей (или несколько "копий" одного и того же пользователя) могут работать параллельно на разных виртуальных консолях, они могут параллельно зарегистрироваться и работать параллельно в разных графических средах. Обычно в стандартно настроенной Linux-системе можно организовать не менее трёх графических консолей, работающих одновременно. Переключаться между ними можно при помощи сочетаний клавиш `Ctrl+Alt+F7`--`Ctrl+Alt+F9`.

Чтобы переключиться из графического режима в одну из текстовых виртуальных консолей, достаточно нажать комбинацию клавиш `Ctrl+Alt+FN`, где "`N`" -- номер необходимой виртуальной консоли.

Простейшие команды

Работа в Linux при помощи командной строки напоминает *диалог* с системой: пользователь вводит команды (реплики), получая от системы ответные реплики, содержащие сведения о произведённых операциях, дополнительные вопросы к пользователю, сообщения об ошибках или просто молчаливое согласие выполнить следующую команду(7).

Простейшая команда в Linux состоит из одного "слова" -- названия программы, которую необходимо выполнить. Одну такую команду (`passwd`) Мефодий уже использовал для того, чтобы изменить свой пароль. Теперь Мефодий решил вернуться на одну из виртуальных консолей, на которой он зарегистрировался, и попробовать выполнить несколько простых команд.

```
[methody@localhost methody]$ whoami
methody
[methody@localhost methody]$
```

Команда whoami

Название этой команды происходит от английского выражения "Who am I?" ("Кто я?"). В ответ на эту команду система вывела только одно слово: "methody" и завершила свою работу, о чём свидетельствует вновь появившееся **приглашение командной строки**. Программа `whoami` возвращает название учётной записи того пользователя, от имени которого она была выполнена. Эта команда полезна в системах, в которых работает много разных пользователей, чтобы не воспользоваться по ошибке чужой учётной записью. Однако, в приглашении командной строки зачастую указывается имя пользователя (как и в наших примерах), поэтому без команды `whoami` можно обойтись. Следующий пример демонстрирует программу, которая выдаст Мефодию уже больше полезной информации: `who` ("Кто").

```
[methody@localhost methody]$ who
methody       tty1         Sep 23 16:31 (localhost)
methody       tty2         Sep 23 17:12 (localhost)
[methody@localhost methody]$
[methody@localhost methody]$ who am i
methody       tty2         Sep 23 17:12 (localhost)
[methody@localhost methody]$
```

Команда who

`who` выводит список пользователей, которые в настоящий момент зарегистрированы в системе (вошли в систему). Данная программа выводит по одной строке на каждого зарегистрированного пользователя: в первой колонке указывается **имя пользователя**, во второй -- "точка входа" в систему, далее следует дата и время регистрации и **имя хоста**. Из выведенной `who` информации можно заключить, что в системе дважды зарегистрирован пользователь `methody`, который сначала зарегистрировался на первой виртуальной консоли (`tty1`), а примерно через сорок минут -- на второй (`tty2`). Конечно, Мефодий и так это знает, однако администратору больших систем, когда пользователи могут зарегистрироваться со многих компьютеров и даже по Сети, программа `who` может быть очень полезна. Могло создаться впечатление, что `who` -- очень умная программа, понимающая английский, но это не так. Из всех английских слов она понимает только сочетание "am i" -- таким способом Мефодий узнал, за какой консолью он сейчас работает.

Ещё одна программа, выдающая информацию о пользователях, работавших в системе в последнее время -- `last`(8). Выводимые этой программой строки напоминают вывод программы `who`, с той разницей, что здесь перечислены и те пользователи, которые уже завершили работу.

```
[methody@localhost methody]$ last
methody  tty2         localhost        Thu Sep 23 17:12   still logged in
methody  tty1         localhost        Thu Sep 23 16:31   still logged in
cacheman ???          localhost        Thu Sep 23 16:15 - 16:17 (00:01)
cacheman ???          localhost        Thu Sep 23 16:08 - 16:08 (00:00)
cyrus    ???          localhost        Thu Sep 23 16:08 - 16:08 (00:00)
reboot   system boot  2.4.26-std-up-al Thu Sep 23 16:03         (04:13)
```

Команда last

В этом примере Мефодий неожиданно обнаружил кроме себя самого неизвестных ему пользователей `cacheman` и `cyrus` -- он совершенно точно знает, что не создавал учётных записей с такими именами. Это **псевдопользователи** (или **системные пользователи**) -- специальные учётные записи, которые используются для работы некоторыми программами. Поскольку эти "пользователи" регистрируются в системе без помощи монитора и клавиатуры, их "точка входа" в систему не определена (во второй колонке записано "`???`"). В выводе программы `last` появляется даже пользователь `reboot` (перезагрузка). В действительности такой учётной записи нет, программа `last` таким способом выводит информацию о том, когда была загружена система.

Выход из системы

В строках, выведенных программой `last`, указан не только момент регистрации пользователя в системе, но и момент завершения работы. Можно представлять Linux как закрытое помещение: чтобы начать работать нужно сначала *войти* в систему (зарегистрироваться, пройти процедуру идентификации), а после того, как работа закончена, нужно из системы *выйти*. В том случае, если в систему вошло несколько пользователей, каждый из них должен выйти, завершив работу, причём совершенно не имеет значения, разные это пользователи или "копии" одного и того же.

Вход пользователя в систему означает, что нужно принимать и выполнять его команды и возвращать ему отчёты о выполненных действиях, например, предоставив ему интерфейс командной строки. Выход означает, что работа от имени данного пользователя завершена и

более не следует принимать от него команды. Весь процесс взаимодействия пользователя с системой с момента регистрации до выхода называется **сеансом работы**. Причём если пользователь входит в систему несколько раз под одним и тем же именем, ему будут доступны несколько *разных* сеансов работы, не связанных между собой.

В наших примерах Мефодий зарегистрирован в системе дважды: на первой и второй виртуальных консолях. Чтобы завершить работу на любой из них, ему достаточно в соответствующей командной строке набрать команду `logout`.

```
[methody@localhost methody]$ logout
Welcome to Some Linux / tty1
localhost login:
```

Команда logout

В ответ на эту команду вместо очередного приглашения командной строки возобновляется приглашение к регистрации в системе. На данной виртуальной консоли работа с Мефодием завершена, и теперь здесь снова может зарегистрироваться любой пользователь.

Есть и другой, ещё более "немногословный" способ сообщить системе, что пользователь хочет завершить текущий сеанс работы. Нажав `Alt+F2` Мефодий попадёт на вторую виртуальную консоль, где всё ещё открыт сеанс для пользователя "`methody`" и нажмёт сочетание клавиш `Ctrl+D`, чтобы прекратить и этот сеанс. Комбинация клавиш `Ctrl+D` приводит не к передаче компьютеру очередного символа, а к закрытию текущего **входного потока данных**. Грубо говоря, командная оболочка вводит команды пользователя с консоли, как если бы она читала их построчно из файла. Нажатие `Ctrl+D` сигнализирует ей о том, что этот "файл" закончился, и теперь ей неоткуда больше считывать команды. Такой способ завершения совершенно аналогичен явному завершению командной оболочки командой `logout`.

---

(1) Вместо формального "зарегистрироваться в системе" обычно используют выражение "войти в систему". Операционная система представляется чем-то вроде замкнутого помещения, внутри которого можно оказаться, только успешно проникнув через "дверь" -- пройдя процедуру регистрации.

(2) Это может оказаться важным, например, в такой ситуации: учётную запись пользователя с именем test удалили из системы, а потом добавили снова. Однако с точки зрения системы это уже другой пользователь, потому что у него другой UID.

(3) Обычно Linux выдаёт нормальным пользователям UID, начиная с "`500`" или "`1000`".

(4) Как правило, численное значение GID в этом случае совпадает со значением UID.

(5) Разнообразие графических интерфейсов Linux гораздо выше, чем, например, в Windows, поэтому составить учебный курс, не ориентируясь специально на ту или иную версию, просто невозможно.

(6) В разных дистрибутивах Linux используется разные версии программы `passwd`, поэтому не всегда она будет столь придирчива, как в дистрибутиве Мефодия.

(7) Реплики в таком диалоге строго чередуются, а собеседники не могут говорить одновременно -- в естественном диалоге так никогда не происходит. Скорее это напоминает диалог в учебнике иностранного языка. Однако и в диалоге с Linux у собеседников есть возможность "перебить" друг друга -- об этом речь пойдёт в последующих лекциях.

(8) В некоторых Linux-системах эта программа может называться `lastlog`.

История

Предыстория

В те далекие времена, когда компьютеры были очень большими, а программы -- очень маленькими, разработка программного обеспечения была неотъемлемой частью разработки компьютера. В результате программное обеспечение было жестко привязано к аппаратному обеспечению конкретной машины, позже -- к серии однотипных машин.

Именно тогда и появились сами термины "аппаратное обеспечение", "программное обеспечение". Простой арифмометр или мясорубка состоит только из "железа", которое работает само по себе, в то время как "железные" элементы любого компьютера, даже правильно собранные, сами по себе не заработают: для работы компьютера необходимы программы. Для функционирования компьютера аппаратное обеспечение должно быть "скреплено" программным.

В те времена, когда все программное обеспечение было составляющей частью компьютера (и не подлежало замене на другое), оно разрабатывалось с той же скоростью и в то же время, что и аппаратное обеспечение. Например, время проектирования ЭВМ Мир, начиная с теоретических разработок и заканчивая эксплуатационным образцом, составило 12 лет. И на протяжении всего этого времени параллельно с аппаратным разрабатывалось программное обеспечение, именно для "скрепления" аппаратуры, а не для решения пользовательских задач. Пользователи для решения своих задач писали программы сами. Ведущей компьютерной фирмой США --- IBM --- использовались аналогичные подходы: создание системы IBM System/360 от начала разработки до начала стабильной работы заняло так же около десяти лет и операционная система OS/360 была неотъемлемой частью этой ЭВМ.

Неотъемлемость программного обеспечения от аппаратного можно наблюдать и в некоторых современных устройствах, например, в неуправляемых свитчах. В них нет операционной системы, тем не менее, в них есть регистры для хранения MAC-адресов, программный код для работы с ними и так далее.

Исторической и идеологической предшественницей GNU/Linux является операционная система UNIX. История UNIX весьма обширна и интересна, но мы заострим внимание лишь на трех вещах, её касающихся.

Разработка для себя

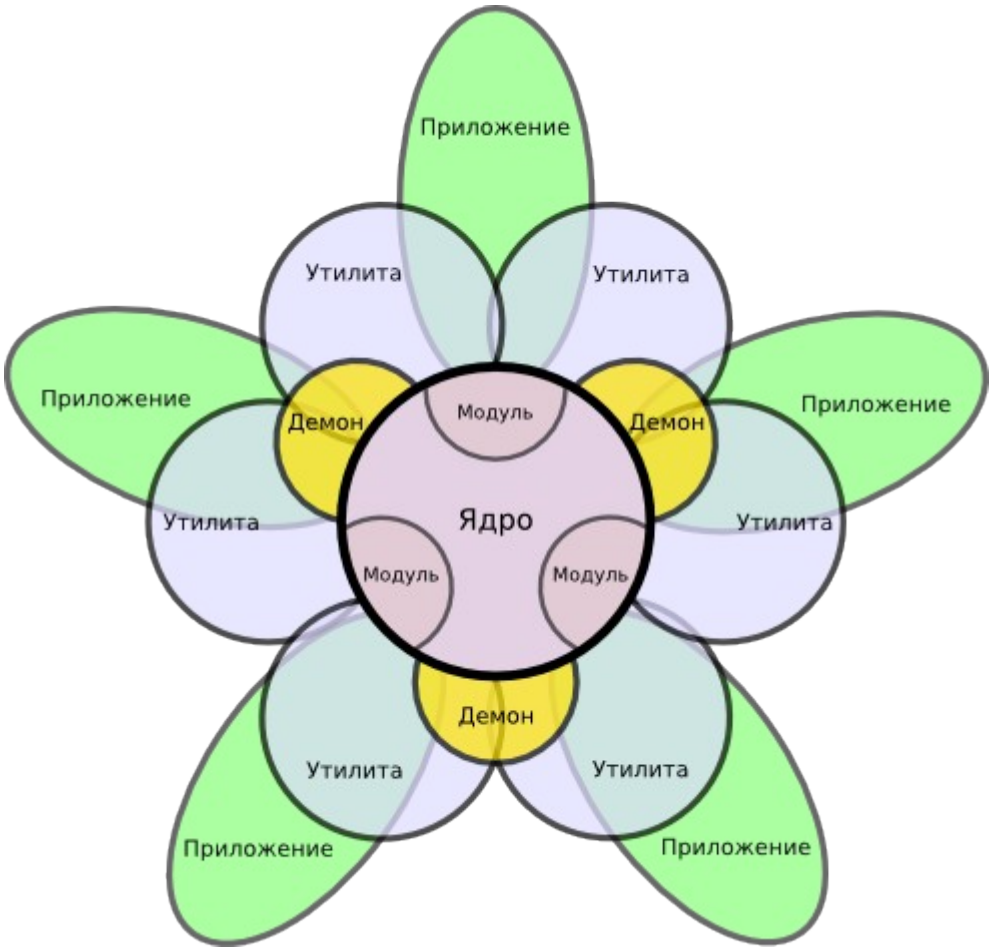
Исторически операционная система UNIX разрабатывалась "для себя". После того, как Bell Labs (исследовательское подразделение AT&T) отказалось от участия в проекте MULTICS, группа исследователей из Bell Labs, задействованная в этом проекте, хотела продолжить работу над созданием операционной системы с разделением времени. Название UNIX предложил Брайан Керниган, по ассоциациям с MULTICS. Предложение купить новый компьютер для проекта было отклонено и поначалу для экспериментов использовался относительно старый PDP-7.

Немного ранее, при работе над Multics, Кен Томпсон создал игру Space Travel, и был ею весьма увлечен, кроме того, он разработал язык B (предтечу языка C), компилятор которого был создан при участии Дениса Ритчи. Для работы Space Travel и был использован компьютер PDP-7 как обладающей неплохим текстовым дисплеем. В ходе работ по переносу Space Travel на PDP-7 была начата разработка утилит, необходимых для удобной разработки игры прямо на PDP-7. В результате этих работ была по сути создана однопользовательская операционная система, названная UNICS (в противоположность MULTICS). В дальнейшем в нее была добавлена многозадачность, и название изменилось на UNIX.

Чтобы на одном компьютере могли одновременно выполняться несколько процессов и

работать несколько пользователей рабочей среде нужно было ядро, отвечающее за грамотное распределение ресурсов --- машинного времени, оперативной памяти, внешних устройств. Уже тогда было очевидно, что такое ядро должно быть обособлено от пользовательских задач, и должно лишь предоставлять возможность разделения ресурсов.

С точки зрения программ, ядро операционной системы --- это всего лишь большая библиотека, предоставляющая функции для управления ресурсами. Ресурс можно заказать, освободить, можно получить отказ в доступе к ресурсу, и т.п. Доступ к программному интерфейсу ядра предоставляется в виде системных вызовов (system calls, 2-ая секция man). Программы, позволяющие воспользоваться функциями ядра называют утилитами. По замыслу разработчиков набор утилит должен реализовывать командный интерфейс ядра на основе программного. Утилиты позволяют манипулировать файлами, производить печать, и т. д.



Кен Томпсон и Денис Ритчи продвинулись в реализации этой концепции. В начале 70-ых задачей начал заниматься целый отдел, началось внедрение. Начальник отдела Дуглас Макилрой увлекался макроязыками и предложил идею конвейера, т.е. потоковую передачу данных между процессами и макроязыка для ее описания. В технологии конвеера отсутствует способ нормальной реализации механизма ветвления, однако, при использовании командной строки механизмы сложнее конвейера могут быть трудными для использования пользователем.

Затем в разработке начал принимать участие известный популяризатор науки Брайан Керниган. Он предложил, во-первых, добавить в язык B некоторые высокоуровневые средства из PL/I, например, структуры. Во-вторых, Керниган предложил переписать операционную систему с PDP-ассемблера на язык C. Идея состояла в том, что при написании большей части операционной системы на неком языке программирования, этой операционной системой потенциально можно оснастить любой компьютер, для которого

существует компилятор использовавшегося языка.

В то время уже существовали языки программирования, такие как PL/I, FORTRAN, ALGOL и LISP. Но они были ориентированы больше на решение пользовательских задач, чем системных. FORTRAN (от *FORmula TRANslator*), первый высокоуровневый язык, был придуман математиками и ориентирован на решение математических задач. ALGOL-60 имел некоторые особенности, не позволявшие создавать нормальные реализации. PL/I только начинал приобретать законченную форму, и также был ориентирован скорее на пользовательские задачи, хотя и содержал много интересных идей. В результате для написания ядра операционных систем использовался машинный язык соответствующей ЭВМ.

Керниган предложил реализовать язык программирования, ориентированный на системные задачи, и достаточно простой для того, чтобы реализация его компилятора на любой платформе не представляла собою сложную задачу, эдакий "переносимый ассемблер". Потенциальным результатом использования подобного подхода была возможность портирования операционной системы на компьютер с принципиально отличающейся архитектурой. В 1972 году ОС Unix была портирована на 32-разрядный Interdata-32 с 16-разрядного PDP-11. Небольшая часть ядра операционной системы, разумеется, реализовывалась и продолжает реализовываться на ассемблере, однако вся логика операционной системы была написана на языке С.

Переносимость программ

Портируемость программных продуктов стала, в определенном смысле, революционным событием. Появилась возможность создать программный продукт системного уровня один раз, и после этого использовать его на различных компьютерах. В случае одинаковой архитектуры достаточно было простого копирования скомпилированного кода, в случае разных --- перекомпиляции. В результате жизненный цикл программного продукта полностью отделился от жизненного цикла компьютера и даже класса компьютеров. Это начало происходить в 70-ые годы. Стало ясно, что производство программных продуктов отличается от производства глиняных горшков: для того, чтобы сделать в два раза больше горшков надо потратить в два раза больше времени, сил, глины; для того, чтобы сделать две копии программы достаточно вызвать утилиту копирования. Стало очевидно, что переносимые программные продукты не являются материальными, поскольку больше не являются частью "программно-аппаратных комплексов", какими были ЭВМ до 70-ых годов, и могут являться новым товаром, если внушить людям необходимость его покупать. Эта идея была сформулирована известным бизнесменом Биллом Гейтсом в середине 70-ых в "Открытом письме любителям".

Коллективная разработка

В результате отделения программного обеспечения от аппаратного выяснилась еще одна вещь: для создания хорошей программы целесообразно привлечь к её разработке разных людей, заинтересованных в немного разных задачах. Например, кто-то пишет программу для форматирования текста для печати на принтере IBM. Благодаря портируемости, эту программу могут использовать еще в десятке организаций. В одной из организация сотрудник хочет напечатать поздравление секретарше шефа в стихотворной форме и добавляет в программу возможности центрирования и использования красивых шрифтов. Суть в том, что пользователи модифицируют программный продукт, приспосабливая его к своим нуждам.

После того, как UNIX получила распространение, в течение порядка десяти лет она развивалась вышеописанным способом. Люди писали программы, на конференциях

обменивались ими, находили ошибки и дорабатывали, снова обменивались. Этот процесс затронул не столько UNIX, разрабатывавшийся Кеном Томпсоном и Денисом Ричи в Bell Labs (тот был производственным продуктом и принадлежал компании AT&T), сколько UNIX-подобную ОС, создававшуюся американскими университетами и получившую название BSD (*Berkeley System Distribution*, по названию самого известного кампуса Университета Калифорнии). Тогда же появилась концепция создания и распространения некоторого блока программ,написанных различными людьми, называющегося *distribution*. Авторы программ распространяли их затем, чтобы другие пользователи подправляли и дорабатывали программы. Такая форма разработки программного продукта хорошо себя зарекомендовала уже тогда, хотя в те времена, чтобы передать копию программы надо было записать её на ленту и либо отдать при личной встрече, либо выслать бандеролью. Программы тогда разрабатывались не слишком многочисленными учеными, которые постоянно встречались и общались на различных конференциях, обмениваясь не только идеями, но и их воплощениями.

Это счастливое время продолжалось до конца 70-ых-начала 80-ых годов, когда стала очевидно, что ОС UNIX и её подобия обладают с точки зрения бизнеса двумя очень важными свойствами:

- для того, чтобы снабдить переносимой ОС новый компьютер/тип компьютеров надо значительно меньше времени, чем тратилось на разработку ОС ранее. Например, полгода вместо пяти лет;
- UNIX проста и технологична в своей архитектуре, что позволяет гибко конструировать всевозможные решения пользовательских задач.

Идея UNIX состояла в том, что система разделялась на ядро, системные утилиты, собственно приложения, которые в основном перерабатывают пользовательскую информацию, и пользуются утилитами для доступа к ядру или сами вызывают системные вызовы. Приложения решают пользовательские задачи, в то время как утилиты отвечают в идеале за какую-то одну четко очерченную функцию. Эта архитектура оказалась достаточно гибкой, чтобы с помощью ее решать практически любую задачу, которая стоил перед пользователем, причем, в том числе, решать силами самого пользователя. Таким образом, UNIX изначально был конструктором для достаточно опытных пользователей.

Лицензионно-правовые аспекты

В 80-ые годы стало выяснятся, что большинство программного кода UNIX-систем имеет правообладателей. Более того, если программа не была написана в свободное от работы время и иного не указано в контракте, правообладателем является работодатель программиста (им мог быть и университет). Многие правообладатели не хотели свободного распространения исходного кода. Это совершенно не согласовывалось с описанным выше академическим стилем создания программ. Разработчики UNIX-систем привыкли показывать свои работы заинтересованным коллегам, которые высказывали мнения, помогали, дополняли. Работа в сообществе сильно отличается от работы в одиночку. Также выяснилось, что многие не задумывались о принадлежности кода, и для большого количества программ установить писались ли они в рабочее время, и кто написал какую часть не представляется возможным. В результате значительная часть кода была признана принадлежащим компании AT&T (владельцу Bell Labs на момент появления UNIX). Кроме того, на основе BSD была создана закрытая коммерческая UNIX-система SunOS.

В результате указанных факторов в 80-ых годах, параллельно с развитием университетских версий, произошел стремительный рост коммерческих UNIX-подобных систем: полное дерево родословной UNIX-систем занимает около 24 листов формата A4.

В начале 80-ых годов стала повышаться информационная связность университетов. В

Америке и крупных исследовательских центрах Европы появился Интернет. Компьютеры объединялись в группы, внутри которых люди могли обмениваться программами уже не посылая бандеролью магнитные ленты. Эффективность команд, которые состояли из заинтересованных людей, находящихся в разных местах, сильно повысилась, появились первые примитивные средства групповой работы.

Таким образом, с одной стороны была группа людей, которая хотела придерживаться академического стиля разработки и создавать свободное ПО. Они предпочитали придерживаться научного подхода к программированию --- ученые проводят исследования, публикуют статьи и исходные тексты программ, и все могут их использовать. При необходимости ученые могут оказывать помощь друг другу. С другой стороны, появились люди, осознавшие, что разработку программного обеспечения можно превратить в промышленную отрасль с большими доходами. Эти люди были убеждены, что производство несвободного и платного ПО можно организовать намного эффективней, чем производство ПО свободного и бесплатного. К рассмотрению этого вопроса мы вернемся позднее.

Стало ясно, что необходимо строго разделить свободное ПО, с которым можно делать что угодно (или почти что угодно) без ответственности перед правообладателем, и несвободное, распространение исходников которого незаконно. На собственном примере это понял одни из самых известных людей в области свободного ПО --- Ричарда Мэтью Столлмана (RMS). В частности, в начале 80-ых он работал в над lisp-машиной (компьютером, имеющим язык lisp в качестве ассемблера), и вдруг выяснилось, что всё им разработанное, в том числе и текстовый редактор Emacs, написанный для личных нужд, принадлежит компании-работодателю, так как было сделано в рабочее время.

В результате, в истории UNIX-систем 80-ые годы известны как UNIX wars, из-за двух конфликтов:

- конфликт между академическими и коммерческими разработчиками;
- конфликт между поставщиками ПО, которые породили множество вариантов UNIX, в большинстве закрытых и несовместимых друг с другом благодаря отсутствию стандартов.

С точки зрения развития операционных систем, 80-ые стали временем стагнации. Десятилетие прошло под лозунгом "компьютер в каждый дом", но при этом технологический прогресс приостановился. На 8-битных и 16-ти битных домашних компьютерах не было даже полноценных операционных систем. Внимание уделялось разработке интерфейсов, повышению *usability* (особенно это отличало Apple Macintosh), но как операционная система Mac OS не был новаторским продуктом: через 15 лет Mac OS X стал базироваться на все том же BSD. Технологии же не могут развиваться без исследований, а исследования не могут существовать без академической структуры.

К концу 80-х возросшая информационная связность позволила сформироваться свободному сообществу вокруг программных продуктов: при совместной работе над проектом людям более не приходилось платить за свой энтузиазм. В конце 80-ых -- начале 90-х в связи с развитием интеренета стали отмирать такие вещи как система UUCP, FIDO, телефонные BBS-системы, в какой-то период не выпускались даже GNU Distributions (подборки свободного ПО). В России, в силу не очень широкого распространения Интернета, всё это использовалось намного дольше.

Вместе с со связностью, возросло и напряжение между свободным и несвободным. Появились случаи серьезного преследования, инициируемого проавообладателями, наиболее известным был иск AT&T к университету Berkeley, разрабатывающем BSD.

Свободное лицензирование программ

При распространении программного продукта правообладатель накладывает на него некоторые условия распространения. Ровно также, как в этих условиях может содержаться запрет на копирование, модификацию, дизассемблирование, в них может быть сказано и примерно следующее:"Программу можно дизассемблировать, распространять, модифицировать, совершать прочие действия, но при дальнейшем распространении нельзя ее закрывать или иным способом делать менее свободной."

После ряда эпизодов Столлман понял, что закрытое программное обеспечение может как нарушать права пользователей, которые не могу его адаптировать для своих нужд, так и разработчиков, которые не имеют права на продукт собственного труда, поскольку он принадлежит их работодателю. Столлман не остановился на формулировании идеи о том, что ПО должно быть свободным, а перешел к действиям --- он изобрел свободное лицензирование программных продуктов и создал с помощью юриста юридически корректную лицензию, воплощающие его идеи, а так же создал фонд, посвященный защите, пропаганде и разработке свободного программного обеспечения. В качестве своей цели RMS выбрал создание свободной UNIX-подобной операционной системы.

Используя американский механизм авторской лицензии, RMS и его коллеги из Free Software Foundation (FSF) добились юридической значимости свободной лицензии General Public License (GPL). Во многих странах GPL либо сама является значимой, либо, как в России, принимается во внимание. Таким образом, разработчики получили возможность распространять свои программные продукты по условиям лицензии, запрещающей делать свободное ПО менее свободным.

В рамках движения FSF было создано большое количество свободных программных продуктов, включая компилятор языка C GCC, однако долгое время не было ядра, лицензированного по GPL. В принципе, существовала теоретическая перелицензировать ядро BSD, но этого так и не произошло. Возможно по причине того, что ядро BSD имело код AT&T и BSD в итоге была втянута в UNIX wars, а возможно из-за каких-то разногласий между RMS и разработчиками BSD.

К началу 90-хы годов у FSF были все основные части свободной операционной системы --- кроме работающего ядра. Один финский (этнически --- шведский) студент, Линус Торвальдс в то время экспериментировал с учебной ОС Minix. ОС Minix распространялась с исходниками, но на тот момент была несвободной. Ее главным автором был академический классик в области построения ОС --- Э. Танненбаум. В ходе этих экспериментов Линус написал собственную маленькую UNIX-подобную ОС. ОС Линуса умещалась на дискете и, по обычаям того времени, была несвободной: её можно было изучать, но распространение допускалось только в университетах.

Главной ценностью работы Линуса было работоспособное ядро, хотя в тот момент и с очень скромными возможностями. Благодаря участию энтузиастов по всему миру, ядро росло и развивалось, а в 1992-м году для его кода была выбрана лицензия GPL. Таким образом. В 1993-м года Патрик Фолькердинк в качестве дипломной работы объединил свободное ядро Linux и свободные утилиты GNU, создав, всего за несколько месяцев, дистрибутив операционной системы, то есть нечто, что после установки чего на компьютере появляется операционная система. Это было беспрецедентное явление, так как ранее создание полноценных UNIX-подобных операционных систем считалось прерогативой крупных компаний. Благодаря активности заинтересованных в свободной ОС пользователей Патрик продолжал и продолжает делать релизы своего дистрибутива, названного им Slackware.

Как только процесс создания операционной системы стал занимать не несколько лет, а несколько месяцев, начали появляться и другие дистрибутивы --- Debian, SuSe, RedHat. Некоторые люди сочли, что создание дистрибутива это новая технология, позволяющая

продавать услуги. В течение 90-ых годов количество дистрибутивов, получивших название "операционная система GNU/Linux" неуклонно росло. Итак, дистрибутив --- это не свободные программы, и, тем более, не ядро. Дистрибутив --- это сделанная из разрозненного набора компонент операционная система с уникальными свойствами. Например, Ubuntu и ALT Linux далеко не одинаковы, хотя и то и то является операционной системой GNU/Linux.

Принципы человеко-машинного взаимодействия в GNU/Linux

Парадигмы GNU/Linux

С точки зрения изучения системы GNU/Linux, излишняя привязка к конкретной графической среде весьма сомнительна по следующим причинам:

- во-первых, графическая оболочка сама по себе не дает понимания, "что такое Линукс" как операционная система;
- во-вторых, может наступить момент, когда придется переходить на другую графическую среду, с другими принципами организации: только самых распространенных сред --- три (KDE, Gnome, XFCE);
- в-третьих, существует целый класс задач системного администрирования, решаемых с интенсивным использованием командной строки.

Ответим теперь на вопрос: что же такое GNU/Linux (или, неформально, "Линукс")? Мы уже отметили, что это, в первую очередь, свободный программный продукт, разрабатываемый множеством сообществ разработчиков. Однако, это ответ с точки зрения процесса разработки и он неполон. Ответ на этот вопрос можно получить, рассмотрев базовые принципы человеко-машинного взаимодействия в GNU/Linux. Человек должен не просто управлять компьютером, а обладать инструментом для решения (и изобретения решения) любой задачи. Сформулируем теперь ключевое утверждение: GNU/Linux --- это свободная операционная система, то есть комплекс программных средств, обеспечивающий унификацию, разделение и разграничение ресурсов для пользовательских программ, которая базируется на следующих трех принципах.

1. **"Все --- файл"**. Для управления операционной системой и GNU/Linux и ее системными сервисами используются файлы. Таким образом, одним из центральных объектов нашего изучения становится файловая система (ФС). Заметим, что этот термин может означать как способ хранения файлов на диске (физическая организация), так и способ представления файлов для пользователя (логическая организация). Речь в данном случае идет, разумеется, о втором из значений. С достаточной степенью точности можно считать, что с точки зрения пользователя файловая система состоит из выстроенных в древовидную структуру каталогов и файлов. Организация файловой системы в GNU/Linux стандартизована и вполне умопостижима --- разобраться, что где лежит и для чего нужно обыкновенно не составляет труда.
2. **"Все --- текст"**. Файлы, требующие ручной или автоматической модификации, --- текстовые. С точки зрения пользователя, структура файловой системы (являющаяся некоторой проекцией всей ОС) представляется в виде дерева файлов, причем в файлах конфигурации информация хранится в пригодном для чтения и изменения человеком текстом виде. Файлы, не предназначенные для непосредственого чтения и модификации пользователем, могут быть бинарными, например к ним относятся программы в бинарном и библиотеки в откомпилированном виде.

3. Управление системой есть работа со набором специальных инструментов, манипулирующим текстовыми файлами (строго говоря, файлами и текстом вообще). Сам инструмент также не выходит за рамки парадигмы "файл --- текст": человек и машина обмениваются текстами, в результате чего модифицируются файлы. Следствием данного принципа является то, что основным интерфейсом управления GNU/Linux является **интерфейс командной строки**. Понятно, что он очень хорошо подходит для описанной парадигмы: пользователь набирает текстовые команды, компьютер команды интерпретирует, исполняет и в качестве результата выдает обратно текст.

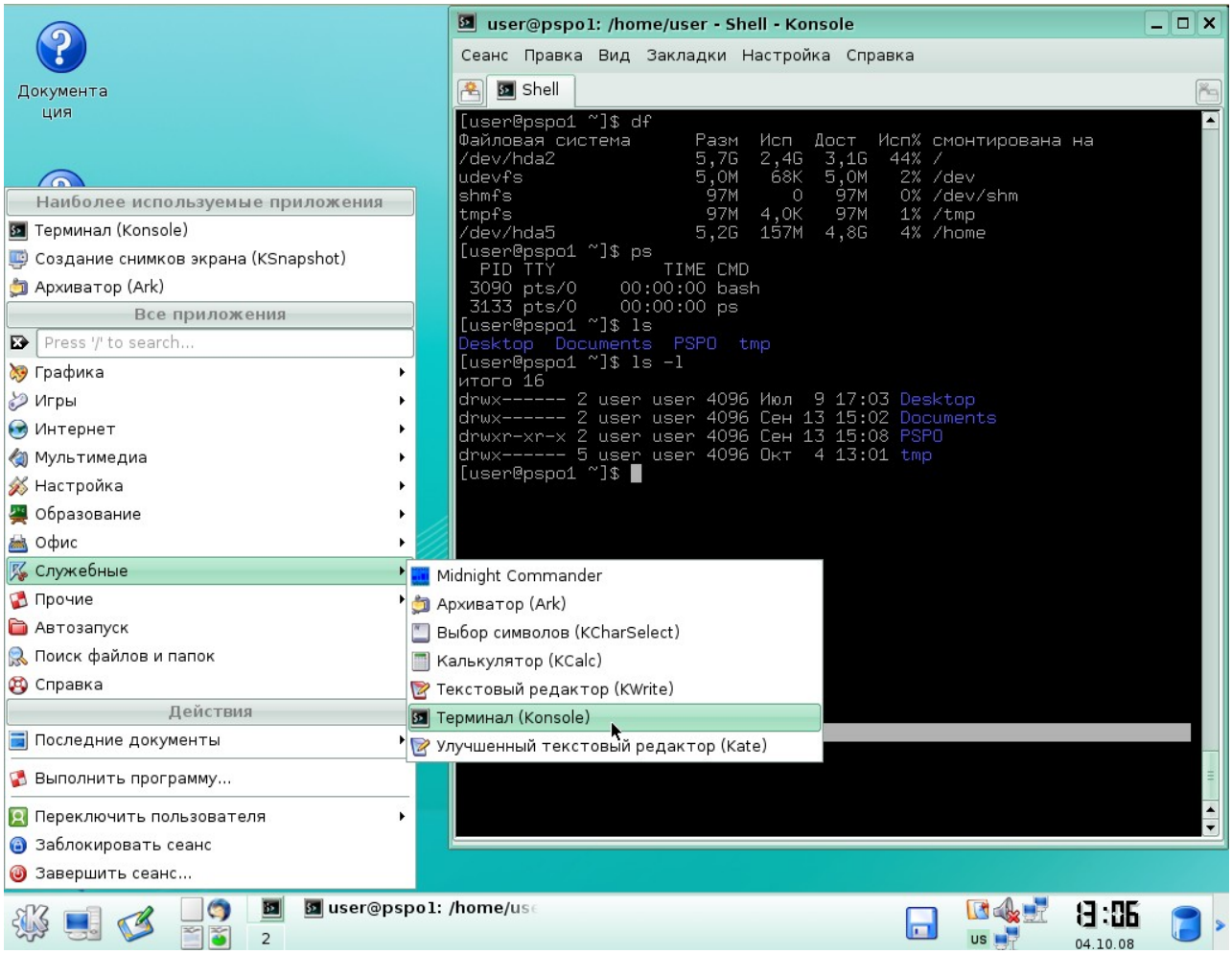
Следует отметить, что все сказанное выше относится к любой Unix-подобной операционной системе вообще, поскольку известная расшифровка аббревиатуры GNU как *GNU is Not Unix* касалась только несвободной природы UNIX в 80-ые годы, но не ее принципов как операционной системы.

Безусловно, существуют выходящие за рамки описанной парадигмы пользовательские задачи. Заметим, однако, что речь в данном случае идет об управлении операционной системой как таковой. Чтобы пользоваться компьютером как точкой запуска интернет-браузера, аудиоплеера и почтового клиента, не обязательно разбираться во всех тонкостях: все нужные приложения легко найти в меню графической среды пользователя. Если же мы захотим изучать GNU/Linux как операционную систему, то разумно пользоваться следующей схемой. Допустим, мы хотим решить какую-то задачу управления системой, а инструментов перед собой видим недостаточно: у нас есть микроскоп, а нам нужно забить девятнадцать гвоздей. Не следует пытаться решить задачу имеющимися инструментами: как максимум, можно один гвоздь забить микроскопом, чтобы убедиться в нецелесообразности такого подхода. Вместо этого имеет смысл изучить содержимое ящика с инструментами, выбрать из этих инструментов подходящий (молоток) и в дальнейшем пользоваться для забивания гвоздей именно им.

Иными словами, если решение задачи представляется малоэффективным, то это вероятно значит, что используется плохо подходящий инструмент: нужный инструмент попросту не найден или же недостаточно освоен. GNU/Linux предоставляет очень богатый инструментарий, позволяющий пользователю при необходимости объединять имеющиеся инструменты в более сложные. Чтение в таких ситуациях документации позволяет накопить багаж знаний, достаточных для решения всех своих, в том числе довольно изощренных, задач.

Основы использования командной строки

Расскажем теперь об интерфейсе командной строки чуть более формально. Этот интерфейс на самом деле предоставляется специальной программой --- интерпретатором командной строки, называемого также командной оболочкой или "шеллом" (англ. *unix shell*). Другими словами, это не само ядро Linux с нами "разговаривает", а специальный пользовательский процесс --- один из запускаемых при входе пользователя в систему. По умолчанию это запущенный файл `/bin/bash`. Чем занимается эта программа? Она читает то, что мы вводим с клавиатуры, анализирует, выполняет соответствующие задачи и выводит результаты --- как свои, так и других программ --- в виде текста. При использовании графической среды для запуска командного интерпретатора следует выбрать пункт *Терминал* из меню. При использовании среды KDE (Юниор, Мастер) это выглядит так:



Команды для выполнения командной оболочкой могут как вводится с клавиатуры, так и браться из файла, называемого командным сценарием или "скриптом" (англ. *shell script*). Команды не обязательно исполняются последовательно --- на самом деле командная оболочка является интерпретатором специализированного языка программирования! Отметим, что данная парадигма реализуется не только командной оболочкой: подобный интерфейс имеют, скажем, интерпретатор языка программирования Python, клиент базы данных MySQL и другие программы.

Отдаваемые командному интерпретатору команды обыкновенно вводятся построчно. Каждая строка разбивается на слова в соответствии с простым правилом: последовательность любых символов, идущих подряд и не являющихся разделителями --- это слово, а последовательность разделителей --- промежуток между словами. Разделителями являются пробел, символ табуляции и символ перевода строки, причем последний актуален в качестве разделителя при использовании файлов сценариев, а не в случае ввода с клавиатуры.

Итак, введенная строка разбита на слова. Первое из этих слов считается командой, а все остальные --- параметрами этой команды, пока не встретится слово, являющееся признаком конца команды текущей команды и начала новой или строка не закончится. Откуда взялось такое соглашение? Конечно, можно было на каждый случай жизни придумать специальную команду, но понятно, что слова при таком подходе быстро закончились бы и команды в конце концов стали бы выглядеть довольно странно. Запомнить такой букет было бы, вероятно, совершенно невозможно. Поэтому при работе с командной строкой применяется следующий принцип: команда (обычно это имя утилиты, но об этом далее) решает ту или иную пользовательскую подзадачу, а все параметры этой подзадачи (имена обрабатываемых файлов, нюансы работы) передаются при вводе команды в виде слов.

Вначале введем некоторые общепринятые соглашения. Команда следует после символа "$",

который пользователь в видит в конце приглашения к вводу при использовании командного интерпретатора. Например, запись

```
$ ls
```

означает, что следует ввести `ls` и нажать Enter. Если же для выполнения команды должно происходить от лица суперпользователя (root), то перед командной будет выводится символ "#", которым заканчивается приглашение к вводу для пользователя root. Таким образом, команды

```
$ su -
# service network restart
```

означают выполнение команды переключения пользователей `su` с параметром "-" (система запросит пароль суперпользователя) и выполнение команды `service` с параметрами `network` и `restart` с праваими суперпользователя.

Рассмотрим следующую команду:

```
$ #script -t my_script 2> my_script.time
```

где "$", как уже было сказано, не часть команды, а приглашение ввода. Это пример того, как люди используют список введенных ими команд ("историю") в качестве записной книжки. Дело в том, что введенная в начале строки решетка является специальным символом, обозначающим комментарий. Таким образом, команда была лишь введена, но не выполнена, однако при этом она попала в историю введенных команд (эту историю, разумеется, можно просматривать, например клавишами "вверх" и "вниз"). Представим теперь, что символа комментария в начале строки не стоит:

```
$ script -t my_script 2> my_script.time
```

Каким образом будет разобрана такая команда? Интерпретатор выделит имя команды ("script") и два параметра ("-t" и "my_script"). Слово "2>" в данном случае параметром команды не является: оно содержит символ "больше" (">") и является указанием интерпретатору выполнять перенаправление вывода сообщений об ошибках в файл `my_script.time`. К моменту запуска команды это слово будет уже обработано интерпретатором (можно считать, что с точки зрения команды script соответствующая группа символов просто исчезнет).

Фактически, диалог пользователя и системы при помощи командной строки происходит следующим образом. Пользователь набирает команду в командной строке, а интерпетатор эту строку обрабатывает. Обратим внимание на три этапа обработки:

- Командный интерпретатор определяет, что за команда была введена (в данном случае это `script`). В командном интерпретаторе есть небольшой набор встроенных (*built-in*) команд --- чтобы их выполнить, не нужно запускать отдельных программ (хорошим примером является команда `cd`, изменяющая текущий каталог). Если команда является встроенной, то она будет выполнена самим интерпретатором, в противном же случае он по специальному алгоритму будет вначале искать в файловой системе программу с таким именем.
- Командный интерпретатор просматривает строку и ищет в ней специальные управляющие символы. Ранее, когда шла речь об обыкновенных символах и разделителях, мы намеренно допустили некоторую неточность. На самом деле есть и третий тип символов --- управляющие. После того, как интерпретатор находит управляющие символы, он их обрабатывает в соответствии с их значением. В данном случае стандартный вывод (поток) ошибок утилиты `script` перенаправляется не на

терминал, а в файл (сама утилита script этого даже не заметит, если не будет специально "присматриваться").

- После того, как вывод будет перенаправлен, этот фрагмент строки будет считаться обработанным ("удалится"), а утилита `script` будет запущена с двумя параметрами.

Чем же занимается утилита script? Оказывается, она протоколирует все действия, производимые пользователем: записывает в специальный файл вводимые команды и получаемые результаты. Именно эта утилита и использовалась для написания наших примеров.

Заметим теперь, что передаваемые команде параметры обычно логически разбиваются на две группы --- ключи и содержательные параметры. Содержательные параметры --- это имена объектов, строки, специальная информация. Ключи же --- это особого вида параметры, начинающиеся обычно с дефиса ("-"). В силу этого в unix-подобных системах не принято начинать названия файлов с дефиса.

Наличие в командной строке ключей означает, что команда должна выполняться с некоторыми нюансами и изменениями. Ключ `-t` в нашем случае заставляет утилиту `script` дополнительно выводить время (`-t` --- time), содержательный же параметр `my_script` --- это имя файла, с которым, согласно нашей команде, утилита должна работать. Следует отметить, что командный интерпретатор в данном случае ничего не знает о смысле этих ключей, для него это просто две строки.

Будем теперь рассматривать вместо команды `script` команду `ls` (от англ. *list*), показывающую список файлов в текущем, если не указано иного, каталоге. При выполнении некоторых специальных условий она так же выделяет объекты различных типов различными цветами, но не всегда. Например, если вызвать команду `ls` не просто как `ls`, а как `/bin/ls` то выделения цветом не произойдет:

```
$ /bin/ls
Dir1 file1
```

Здесь `/bin/ls` --- это полный путь к соответствующему файлу в дереве каталогов файловой системы, а строка "Dir1 file1" --- это вывод команды `/bin/ls`, который видит пользователь.

Обратим внимание, что отличить имя файла от имени каталога в выводе ls без раскраски невозможно. Можно, однако, указать ключ `-F`, который заставит ls после имен каталогов выводить символ "/". Это типичный пример использования ключа, который модифицирует работу программы. Передаваемый параметр не соответствует никакому объекту: это не файл и не имя. С этим ключом, однако, поведение программы несколько изменяется:

```
$ ls -F
Dir1/  file1
```

Если указать ключ `-s`, то мы увидим и размер объектов с точки зрения физической организации файлвовой системы. Каталог в данном случае имеет размер 4 блока, а файл --- 0 блоков (он пуст).

```
$ ls -F -s
итого 4
4 Dir1/  0 file1
```

Для ключей по возможности соблюдается принцип аббревиативности: вместо полных названий используется одна буква, с названием как-либо связанная. Для ключа -F эта связь неочевидна (от слова *classiFy*), а вот -s означает size (размер). Принцип аббревиативности нужен вот зачем: когда используется сразу несколько ключей, появляется возможность уменьшить количество набираемых символов. Например, команда `ls -F -s` и так не

слишком длинная, однако ее можно еще сократить, поскольку однобуквенные ключи могут прилипать друг к другу:

```
$ ls -Fs
итого 4
4 Dir1/  0 file1
```

Мы ставим один общий дефис, а дальше перечисляем все однобуквенные ключи. Достоинство однобуквенных ключей --- их быстро набирать. Недостатков у них два. Во-первых, не всегда легко запомнить значения всех ключей (смысл сокращений). Отметим, что используются в ключах как маленькие, так и большие буквы. Так, ls с ключом `-a` (англ. *all*) показывает все объекты в текущем каталоге:

```
$ ls -a
.  ..  Dir1  file1  .file1
```

Как видно, по-умолчанию утилита `ls` не показывает объекты с именами, начинающимися с точки. Здесь "." и ".." --- ссылки на каталог верхнего уровня и на текущий каталог. Ключ же -A (*almost all*) заставит вывести "почти все" объекты: будут пропущены "." и ".." (они есть в любом каталоге, и информация о них обычно не нужна):

```
$ ls -A
Dir1  file1  .file1
```

Второй недостаток однобуквенных ключей состоит в том, что алфавит рано или поздно заканчивается и букв начинает не хватать. Когда ключей много или используются некоторые из них редко, то применяют полнословную нотацию. Ключи в этом случае начинаются с двух дефисов, после которых идет полное название параметра (при этом мы не отступаем от правила, предписывающего ключам начинаться с дефиса). Чаще всего однобуквенные ключи имеют полнословные эквиваленты:

```
$ ls --all
.  ..  Dir1  file1  .file1
$ ls --almost-all
Dir1  file1  .file1
```

Важно понимать, что описанные принципы работы с ключами --- это всего лишь соглашения, которые, увы, соблюдаются не всеми и не всегда. Никто не мешает создать программу, с ключами не работающую вовсе или использующую другие принципы их написания. Несколько таких программ (`dd`, `ps`, `tar`) были созданы давно, когда этого соглашения не было, а потому передаваемые им параметры выглядят несколько иначе.