ТЕМА 4.3. UNIX-ПОДОБНЫЕ И ДРУГИЕ POSIX-СОВМЕСТИМЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

В данной теме рассматриваются следующие вопросы:

- Общая характеристика семейства операционных систем UNIX
- Стандарты UNIX.
- История создания UNIX.
- Основные понятия системы UNIX.
- Пользователи системы, атрибуты пользователя.
- Файловая структура ОС.
- Основные свойства UNIX.
- Мобильность операционных систем.
- Методы инсталляции и настройки ОС.
- Особенности процессов.
- Сигналы.
- Обработка сигналов.
- Неименованные каналы.
- Именованные каналы.
- Архитектура ОС UNIX.
- Принципы организации и структура ОС.
- Ядро и основные компоненты ОС.
- iOS.
- ОС Android.

Лекции – 2 часа, лабораторные занятия – 2 часа, самостоятельная работа – 2 часа.

Экзаменационные вопросы по теме:

- Основные понятия системы UNIX. Пользователи системы, атрибуты пользователя. Файловая структура ОС.
- ОС UNIX: особенности процессов, сигналы, обработка сигналов.

4.1.1. Общая характеристика семейства операционных систем UNIX

Операционная система UNIX всегда была интерактивной системой, разработанной для одновременной поддержки множества процессов и множества пользователей. Она была разработана программистами и для программистов — чтобы использовать ее в такой среде, в которой большинство пользователей достаточно опытны и занимаются проектами (часто довольно сложными) разработки программного обеспечения. Во многих случаях большое количество программистов активно работает над созданием общей системы, поэтому в операционной системе UNIX есть большое количество средств, позволяющих людям работать вместе и управлять совместным использованием информации. Очевидно, что модель группы опытных программистов, совместно работающих над созданием сложного программного обеспечения, существенно отличается от модели одного начинающего пользователя, сидящего за персональным компьютером в текстовом процессоре, и это отличие отражается в операционной системе UNIX от начала до конца. Совершенно естественно, что Linux унаследовал многие из этих установок, даже несмотря на то что первая версия предназначалась для персонального компьютера [1].

Чего действительно хотят от операционной системы хорошие программисты? Прежде всего, большинство хотело бы, чтобы их система была простой, элегантной и совместимой.

Другие свойства, которые, как правило, опытные программисты желают видеть в операционной системе, — это мощь и гибкость. Это означает, что в системе должно быть небольшое количество базовых элементов, которые можно комбинировать, чтобы приспособить их для конкретного приложения.

Наконец, у большинства программистов есть сильная неприязнь к бесполезной избыточности. Зачем писать **copy**, когда вполне достаточно **cp**, чтобы однозначно выразить желаемое? Это же пустая трата драгоценного времени.

Первая система Unix была разработана в подразделении Bell Labs компании AT&T. С тех пор было создано большое количество различных Unix-систем [2].

Основное отличие Unix-подобных систем от других операционных систем заключается в том, что это изначально многопользовательские многозадачные системы. То есть в один и тот же момент времени сразу множество людей может выполнять множество вычислительных задач (процессов).

Вторая колоссальная заслуга Unix в её мультиплатформенности. Ядро системы написано таким образом, что его легко можно приспособить практически под любой микропроцессор.

Unix имеет и другие характерные особенности:

- использование простых текстовых файлов для настройки и управления системой;
- широкое применение утилит, запускаемых из командной строки;
- взаимодействие с пользователем посредством виртуального устройства — терминала;
- представление физических и виртуальных устройств и некоторых средств межпроцессного взаимодействия в виде файлов;
- использование конвейеров из нескольких программ, каждая из которых выполняет одну задачу.

Философия Unix

Дуглас Макилрой (Douglas McIlroy), изобретатель каналов UNIX и один из основателей традиции UNIX, обобщил философию следующим образом [2]:

«Философия UNIX гласит:

- Пишите программы, которые делают что-то одно и делают это хорошо.
- Пишите программы, которые бы работали вместе.
- Пишите программы, которые бы поддерживали текстовые потоки, поскольку это универсальный интерфейс».

Обычно эти высказывания сводятся к одному «Делайте что-то одно, но делайте это хорошо».

Из этих трёх принципов только третий является специфичным для UNIX, хотя разработчики UNIX чаще других акцентируют внимание на всех трёх принципах.

В 1994 году Майк Ганцарз (Mike Gancarz) объединил свой опыт работы в UNIX с высказываниями из прений, в которых он участвовал со своими приятелями программистами и людьми из других областей деятельности, так или иначе зависящих от UNIX, для создания Философии UNIX, которая сводится к 9 основным принципам:

1. Простые решения красивы (Small is Beautiful).
2. Пусть каждая программа делает что-то одно, но хорошо.
3. Стройте прототип программы как можно раньше.
4. Предпочитайте переносимость эффективности.
5. Храните данные в простых текстовых файлах.
6. Извлекайте пользу из уже существующих программных решений.
7. Используйте скриптовые языки для уменьшения трудозатрат и улучшения переносимости.
8. Избегайте пользовательских интерфейсов, ограничивающих возможности пользователя по взаимодействию с системой.
9. Делайте каждую программу «фильтром».

В целом можно сказать, что Unix-подобные ОС делались программистами для программистов. Интерфейс не всегда понятен и доступен, требует изучения. Предполагается, что человек должен потратить время, чтобы эффективно пользоваться этими инструментами.

4.3.2. Стандарты UNIX

К концу 1980-х широкое распространение получили две различные и в чем-то несовместимые версии системы UNIX: 4.3BSD и System V Release 3. Кроме того, практически каждый производитель добавлял собственные нестандартные усовершенствования. Этот раскол в мире UNIX вместе с тем фактом, что стандарта на формат двоичных программ не было, сильно сдерживал коммерческий успех операционной системы UNIX, поскольку производители программного обеспечения не могли написать пакет программ для системы UNIX так, чтобы он мог работать на любой системе UNIX (как это делалось, например, в системе MS-DOS). Вначале все попытки стандартизации системы UNIX проваливались. Например, корпорация AT&T выпустила стандарт SVID (System V Interface Definition — описание интерфейса UNIX System V), в котором определялись все системные вызовы, форматы файлов и т. д. Этот документ был попыткой построить в одну шеренгу всех производителей System V, но он не оказал никакого влияния на вражеский лагерь (BSD), который просто проигнорировал его [1].

Первая серьезная попытка примирить два варианта системы UNIX была предпринята при содействии Совета по стандартам при Институте инженеров по электротехнике и электронике (IEEE Standard Boards), глубокоуважаемой и, что самое важное, нейтральной организации. В этой работе приняли участие сотни людей из области промышленности, академических и правительственных организаций. Коллективное название этого проекта — POSIX. Первые три буквы этого сокращения означали Portable

Operating System — переносимая операционная система. Буквы IX в конце слова были добавлены, чтобы имя проекта выглядело юниксоподобно.

После большого количества высказанных аргументов и контраргументов, опровержений и опровергнутых опровержений комитет POSIX выработал стандарт, известный как 1003.1. Этот стандарт определяет набор библиотечных процедур, которые должна обеспечивать каждая соответствующая данному стандарту система UNIX. Большая часть этих процедур делает системный вызов, но некоторые из них могут быть реализованы вне ядра. Типичными процедурами являются **open**, **read**, и **fork**. Идея стандарта POSIX заключается в том, что производитель программного обеспечения, который при написании программы использует только описанные в стандарте 1003.1 процедуры, может быть уверен, что его программа будет работать на любой системе UNIX, соответствующей данному стандарту.

Комитет IEEE взял за основу пересечение множеств функциональных возможностей System V и BSD. То есть дело обстояло примерно так: если какое-либо свойство присутствовало как в System V, так и в BSD, то оно включалось в стандарт, в противном случае это свойство в стандарт не включалось. Документ 1003.1 был написан так, чтобы как разработчики операционной системы, так и создатели программного обеспечения были способны его понять, что также было ново в мире стандартов, хотя в настоящее время уже ведется работа по исправлению этого нестандартного для стандартов свойства.

Несмотря на то что стандарт 1003.1 описывает только системные вызовы, принят также ряд сопутствующих документов, которые стандартизируют потоки, утилиты, сетевое программное обеспечение и многие другие функции системы UNIX. Кроме того, язык C также был стандартизирован Национальным институтом стандартизации США (ANSI) и Международной организацией по стандартизации ISO.

Некоторые примеры стандартизации [2]:

- **C API** расширяет ANSI C такими вещами, как сеть, управление процессами, потоками, файловый ввод-вывод, регулярные выражения.... Реализован в библиотеке glibc, которая в большинстве случаев оборачивает системные вызовы.
- **Язык командной оболочки** (шелл)
- **Переменные окружения** (HOME, PATH, …)
- **Коды выхода программ**. ANSI C утверждает, что 0 или EXIT_SUCCESS — успешный выход, EXIT_FAILURE для неудачи, остальные — в зависимости от реализации. POSIX определяет дополнительные коды, например, 126 (command found but not executable).
- **Регулярные выражения**. Два типа: BRE (Basic) и ERE (Extended). Basic считается устаревшим и оставлен для совместимости.
- **Структура каталогов** (структура каталогов некоторых систем, в частности, GNU/Linux, определена в стандарте Filesystem Hierarchy Standard).

4.3.3. История создания UNIX

Одной из операционных систем второго поколения была MULTICS, совместно созданная MIT, Bell Labs и General Electrics. Кен Томпсон (Ken Thompson) написал на ассемблере усеченный вариант системы MULTICS для компьютера PDP-7. Брайан Керниган (Brian Kernighan), как-то в шутку назвал эту систему UNICS (UNiplexed Information and Computing Service — примитивная информационная и вычислительная служба). Несмотря на все каламбуры и шутки, эта кличка прочно пристала к новой системе, хотя написание этого слова позднее превратилось в UNIX [1].

Вскоре к Кену Томпсону присоединился Деннис Ритчи (Dennis Ritchie), а чуть позднее и весь его отдел. К этому времени относятся два технологических усовершенствования.

Во-первых, система UNIX была перенесена с устаревшей машины PDP-7 на гораздо более современный компьютер PDP-11/20, а позднее на PDP-11/45 и PDP-11/70. Второе усовершенствование касалось языка, на котором была написана операционная система UNIX. Ритчи разработал язык, ставший преемником языка B, который, получил название C, и написал для него прекрасный компилятор. Томпсон и Ритчи совместно переписали UNIX на языке C. Язык C оказался как раз тем языком, который и был нужен в то время, и с тех пор он сохраняет лидирующие позиции в области системного программирования.

Операционная система UNIX быстро завоевала популярность в университетах. — и не в последнюю очередь благодаря тому, что система поставлялась с полными исходными кодами, поэтому новые владельцы системы могли без конца подправлять и совершенствовать ее (и они делали это). Операционной системе UNIX было посвящено множество научных симпозиумов, на них докладчики рассказывали о тех неизвестных ошибках в ядре, которые им удалось обнаружить и исправить. В результате всех этих событий новые идеи и усовершенствования системы распространялись с огромной скоростью.

Version 7 стала первой переносимой версией операционной системы UNIX (она работала как на машинах PDP-11, так и на Interdata 8/32). Эта версия системы состояла уже из 18 800 строк на языке C и 2100 строк на ассемблере. На Version 7 выросло целое поколение студентов, которые, закончив свои учебные заведения и начав работу в промышленности, содействовали дальнейшему ее распространению. К середине 1980-х годов операционная система UNIX широко применялась на мини-компьютерах и инженерных рабочих станциях самых различных производителей.

Первым коммерческим вариантом системы UNIX была System III. Ее выход на рынок был не очень успешным, поэтому через год она была заменена улучшенной версией, System V. Агрессивная маркетинговая политика AT&T по продвижению System V какое-то время препятствовала распространению новых версий. Но впоследствии она продала свой связанный с UNIX бизнес, а основные компьютерные компании уже имели лицензии. Появилось большое количество несовместимых между собой версий, в связи с чем возникли стандарты POSIX, описанные в предыдущем разделе.

Университет в Беркли разработал и выпустил улучшенную версию операционной системы UNIX для мини-компьютера PDP-11, названную 1BSD. Затем последовали версии 2BSD, 4BSD. Университет в Беркли также добавил в систему UNIX значительное количество утилит, включая новый редактор vi и новую оболочку csh, компиляторы языков Pascal и Lisp и многое другое. Все эти усовершенствования привели к тому, что многие производители компьютеров (Sun Microsystems, DEC и др.) стали основывать свои версии системы UNIX на Berkeley UNIX, а не на «официальной» версии System V компании AT&T. В результате Berkeley UNIX получила широкое распространение в академических и исследовательских кругах, а также в Министерстве обороны.

В 1987 году была выпущена операционная система MINIX. Эта система состояла из 11 800 строк на языке C и 800 строк кода на ассемблере и была одной из первых юниксоподобных систем, основанной на микроядре. Идея микроядра заключается в том, чтобы реализовать в ядре как можно меньше функций и сделать его надежным и эффективным. Соответственно управление памятью и файловая система были перенесены в процессы пользователя. Уже через несколько месяцев после своего появления система MINIX стала чем-то вроде объекта культа — со своей группой новостей comp.os.minix и более чем 40 000 пользователей. Очень многие пользователи стали сами писать команды и пользовательские программы, так что система MINIX быстро стала продуктом коллективного творчества большого количества пользователей по всему Интернету, послужив прототипом для других коллективных проектов, появившихся позднее. В 1997 году была выпущена версия 2.0 системы MINIX. Теперь базовая система включала в себя сетевое программное обеспечение, и ее размер вырос до 62 200 строк.

Финский студент Линус Торвальдс (Linus Torvalds) решил сам написать еще один клон системы UNIX, который он назвал Linux. Это должна была быть полноценная производственная система, со многими изначально отсутствовавшими в системе MINIX функциями. Первая версия 0.01 операционной системы Linux была выпущена в 1991 году. Она была разработана и собрана на компьютере под управлением MINIX и заимствовала из системы MINIX множество идей, начиная со структуры дерева исходных кодов и заканчивая компоновкой файловой системы. Однако в отличие от микроядерной системы MINIX, Linux была монолитной системой, то есть вся операционная система размещалась в ядре. Размер исходного текста составил 9300 строк на языке C и 950 строк на ассемблере, что приблизительно совпадало с версией MINIX как по размеру, так и по функциональности. Фактически это была переделка системы MINIX — единственной системы, исходный код которой имелся у Торвальдса.

Операционная система Linux быстро росла в размерах и впоследствии развилась в полноценный клон UNIX с виртуальной памятью, более сложной файловой системой и многими другими дополнительными функциями. Хотя изначально система Linux работала только на процессоре Intel 386 (и даже имела встроенный ассемблерный код 386-го процессора в процедурах на языке C), она была быстро перенесена на другие платформы и теперь работает на широком спектре машин — так же, как и UNIX. Следует выделить одно отличие системы Linux от UNIX: она использует многие специальные возможности компилятора **gcc**, поэтому потребуется приложить немало усилий, чтобы откомпилировать ее стандартным ANSI C-компилятором. История операционной системы Linux будет рассмотрена в следующей теме.

Когда мода на Linux начала набирать обороты, она получила поддержку с неожиданной стороны — от корпорации AT&T. В 1992 году университет в Беркли, лишившись финансирования, решил прекратить разработку BSD UNIX на последней версии 4.4BSD (которая впоследствии послужила основой для FreeBSD). Поскольку эта версия по существу не содержала кода AT&T, университет в Беркли выпустил это программное обеспечение с лицензией открытого исходного кода, которая позволяла всем делать все, что угодно, кроме одной вещи — подавать в суд на университет Калифорнии. Контролировавшее систему UNIX подразделение корпорации AT&T отреагировало немедленно, подав в суд на университет Калифорнии и компанию BSDI. Хотя этот спор в конечном итоге удалось урегулировать в досудебном порядке, он не позволял выпустить на рынок FreeBSD в течение долгого периода — достаточного для того, чтобы система Linux успела упрочить свои позиции.

Если бы судебного иска не было, то уже примерно в 1993 году началась бы серьезная борьба между двумя бесплатными версиями системы UNIX, распространяющимися с исходными кодами: царствующим чемпионом — системой BSD (зрелой и устойчивой системой с многочисленными приверженцами в академической среде еще с 1977 года) и энергичным молодым претендентом — системой Linux всего лишь двух лет от роду, но с уже растущим числом последователей среди индивидуальных пользователей. Кто знает, чем обернулась бы эта схватка двух бесплатных версий системы UNIX.

4.3.4. Основные понятия системы UNIX

Существует два основных объекта операционной системы UNIX, с которыми приходится работать пользователю – файлы и процессы. Эти объекты сильно связаны друг с другом, и в целом организация работы с ними как раз и определяет архитектуру операционной системы [3].

Все данные пользователя хранятся в файлах; доступ к периферийным устройствам осуществляется посредством чтения и записи специальных файлов; во время выполнения программы, операционная система считывает исполняемый код из файла в память и передает ему управление.

С другой стороны, вся функциональность операционной системы определяется выполнением соответствующих процессов. В частности, обращение к файлам на диске невозможно, если файловая подсистема операционной системы (совокупность процессов, осуществляющих доступ к файлам) не имеет необходимого для этого кода в памяти.

Если рассматривать более подробно, операционную систему UNIX можно рассматривать как пирамиду (рис. 4.3.1). У основания пирамиды располагается аппаратное обеспечение, состоящее из центрального процессора, памяти, дисков, монитора и клавиатуры, а также других устройств. Операционная система работает на «голом железе». Ее функция заключается в управлении аппаратным обеспечением и предоставлении всем программам интерфейса системных вызовов. Эти системные вызовы позволяют программам пользователя создавать процессы, файлы и прочие ресурсы, а также управлять ими.

Программы делают системные вызовы, помещая аргументы в регистры (или иногда в стек) и выполняя команду эмулированного прерывания для переключения из пользовательского режима в режим ядра. Поскольку на языке С невозможно написать команду эмулированного прерывания, то этим занимается библиотека, в которой есть по одной процедуре на системный вызов. Эти процедуры написаны на ассемблере, но они могут вызываться из языка С. Каждая такая процедура сначала помещает аргументы в нужное место, а затем выполняет команду эмулированного прерывания. Таким образом, чтобы обратиться к системному вызову **read**, программа на языке С должна вызвать библиотечную процедуру **read**. Кстати, в стандарте POSIX определен именно интерфейс библиотечных функций, а не интерфейс системных вызовов. Иначе говоря, стандарт POSIX определяет, какие библиотечные процедуры должна предоставлять соответствующая его требованиям система, каковы их параметры, что они должны делать и какие результаты возвращать. В стандарте даже не упоминаются реальные системные вызовы.

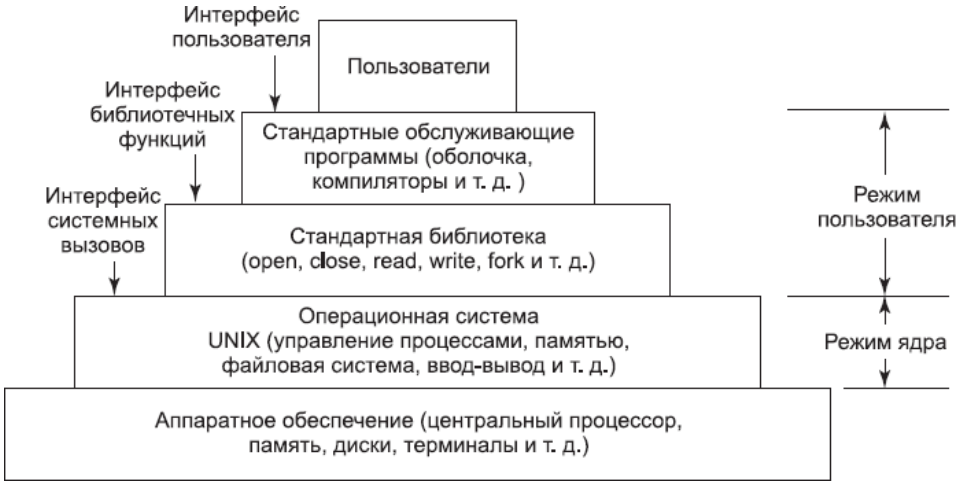

Рис. 4.3.1. Уровни операционной системы Unix

В операционной системе содержится большое количество стандартных программ, некоторые из них указаны в стандарте POSIX 1003.2, тогда как другие могут различаться от версии к версии. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускает пользователь с клавиатуры.

Графические интерфейсы пользователя поддерживает оконная система X Windowing System, которую обычно называют X11 (или просто X). Она определяет обмен и протоколы отображения для управления окнами на растровых дисплеях UNIX-подобных систем. X-сервер является главным компонентом, который управляет такими устройствами, как клавиатура, мышь и экран, и отвечает за перенаправление ввода или

прием вывода от клиентских программ. Реальная среда графического интерфейса пользователя обычно построена поверх библиотеки низкого уровня (xlib), которая содержит функциональность для взаимодействия с X-сервером. Графический интерфейс расширяет базовую функциональность X11, улучшая вид окон, предоставляя кнопки, меню, значки и пр. X-сервер можно запустить вручную из командной строки, но обычно он запускается во время загрузки диспетчером окон, который отображает графический экран входа в систему.

4.3.5. Пользователи системы, атрибуты пользователя

Прежде чем клиент сможет начать работу с ОС UNIX, он должен стать пользователем системы, т.е. получить имя, пароль и ряд других атрибутов. С точки зрения системы пользователь не обязательно человек. Пользователем является объект, который обладает определенными правами и может запускать на выполнение программы и владеть файлами. Пользователями могут быть отдельные клиенты, удаленные компьютеры или группы пользователей с одинаковыми правами и функциями. В системе существует один пользователь, обладающий неограниченными правами это суперпользователь или администратор системы (обычно с именем root) [4].

Каждый пользователь имеет уникальное или регистрационное имя, но система различает пользователей по идентификатору пользователя UID. Идентификаторы также должны быть уникальны. Пользователи являются членами одной или нескольких групп. Группа — список пользователей, имеющих сходные задачи. Принадлежность к группе определяется дополнительными правами, которыми обладают все пользователи группы. Каждая группа имеет уникальное имя, а система различает группы по групповому идентификатору (GID).

Идентификатор пользователя и идентификатор группы определяет, какими правами обладает пользователь в системе. Информация о пользователях обычно хранится в специальном файле: `/etc/passwd`, о группах — `/etc/group`. Этот файл доступен только для чтения. Писать в него может только администратор.

Атрибуты пользователя

Как правило, все атрибуты пользователя хранятся в файле `/etc/passwd`. В конечном итоге, добавление пользователя в систему сводится к внесению в файл `/etc/passwd` соответствующей записи. Однако во многих системах информация о пользователе хранится и в других местах (например, в специальных базах данных), поэтому создание пользователя простым редактированием файла `/etc/passwd` может привести к неправильной регистрации пользователя, а иногда и к нарушениям работы системы. Вместо этого при возможности следует пользоваться специальными утилитами, поставляемыми с системой [5].

Каждая строка файла является записью конкретного пользователя и имеет следующий формат:

```
name:passwd-encod:UID:GID:comments:home-dir:shell
```

— всего семь полей (атрибутов), разделенных двоеточиями.

Рассмотрим подробнее каждый из атрибутов.

name

Регистрационное имя пользователя. Это имя пользователь вводит в ответ на приглашение системы login. Для небольших систем имя пользователя достаточно произвольно. В больших системах, в которых зарегистрированы сотни пользователей, требования уникальности заставляют применять определенные правила выбора имен.

passwd-encod

Пароль пользователя в закодированном виде. Алгоритмы кодирования известны, но они не позволяют декодировать пароль. При входе в систему пароль, который вы набираете, кодируется, и результат сравнивается с полем passwd-encod. В случае совпадения пользователю разрешается войти в систему. Даже в закодированном виде доступность пароля представляет некоторую угрозу для безопасности системы. Поэтому часто пароль хранят в отдельном файле, а в поле passwd-encod ставится символ 'x' (в некоторых системах '!'). Пользователь, в данном поле которого стоит символ '*', никогда не сможет попасть в систему. Дело в том, что алгоритм кодирования не позволяет символу '*' появиться в закодированной строке. Таким образом, совпадение введенного и затем закодированного пароля и '*' невозможно. Обычно такой пароль имеют псевдопользователи.

UID

Идентификатор пользователя является внутренним представлением пользователя в системе. Этот идентификатор наследуется задачами, которые запускает пользователь, и файлами, которые он создает. По этому идентификатору система проверяет пользовательские права (например, при запуске программы или чтении файла). Суперпользователь имеет UID=0, что дает ему неограниченные права в системе.

GID

Определяет идентификатор первичной группы пользователя. Этот идентификатор соответствует идентификатору в файле `/etc/group`, который содержит имя группы и полный список пользователей, являющихся ее членами. Принадлежность пользователя к группе определяет дополнительные права в системе. Группа определяет общие для всех членов права доступа и тем самым обеспечивает возможность совместной работы (например, совместного использования файлов).

comments

Обычно, это полное "реальное" имя пользователя. Это поле может содержать дополнительную информацию, например, телефон или адрес электронной почты. Некоторые программы (например, **`finger`**(1) и почтовые системы) используют это поле.

home-dir

Домашний каталог пользователя. При входе в систему пользователь оказывается в этом каталоге. Как правило, пользователь имеет ограниченные права в других частях файловой системы, но домашний каталог и его подкаталоги определяют область файловой системы, где он является полноправным хозяином.

shell

Имя программы, которую UNIX использует в качестве командного интерпретатора. При входе пользователя в систему UNIX автоматически запустит указанную программу. Обычно это один из стандартных командных интерпретаторов `/bin/sh` (Bourne shell), `/bin/csh` (C shell) или `/bin/ksh` (Korn shell), позволяющих пользователю вводить команды и запускать задачи. В принципе, в этом поле может быть указана любая программа, например, командный интерпретатор с ограниченными функциями (restricted shell), клиент системы управления базой данных или даже редактор. Важно то, что, завершив выполнение этой задачи, пользователь автоматически выйдет из системы. Некоторые системы имеют файл `/etc/shells`, содержащий список программ, которые могут быть использованы в качестве командного интерпретатора.

4.3.6. Файловая структура ОС

Файловую структуру UNIX рассмотрим на примере первых файловых систем Linux (в теме 3.3 уже были рассмотрены некоторые файловые системы).

Первоначально файловой системой в Linux была файловая система MINIX 1. Однако из-за того обстоятельства, что имена файлов были ограничены в ней 14 символами (чтобы поддерживать совместимость с UNIX Version 7), а максимальный размер файла составлял 64 Мбайт (что было даже слишком много для жестких дисков того времени, размер которых составлял 10 Мбайт), интерес к более совершенным файловым системам появился сразу же после начала разработки системы Linux (которая началась примерно через 5 лет после выпуска MINIX 1). Первым улучшением стала файловая система ext, которая позволяла использовать имена файлов длиной 255 символов и размер файлов 2 Гбайт (однако она была медленнее, чем файловая система MINIX 1, так что поиски продолжались еще некоторое время). В итоге была изобретена файловая система ext2 (с длинными именами файлов, большими файлами и более высокой производительностью), которая и стала основной файловой системой. Однако Linux поддерживает несколько десятков файловых систем при помощи уровня виртуальной файловой системы Virtual File System (VFS), описанного в следующем разделе. Когда система Linux собирается, вам предлагается сделать выбор тех файловых систем, которые будут встроены в ядро. Другие можно загружать динамически (как модули) во время выполнения (если будет такая необходимость).

Файл в системе Linux — это последовательность байтов произвольной длины (от 0 до некоторого максимума), содержащая произвольную информацию. Не делается различия между текстовыми (ASCII) файлами, двоичными файлами и любыми другими типами файлов. Значение битов в файле целиком определяется владельцем файла. Системе это безразлично. Имена файлов ограничены 255 символами. В именах файлов разрешается использовать все ASCII-символы, кроме символа NUL, поэтому допустимо даже состоящее из трех символов возврата каретки имя файла (хотя такое имя и не слишком удобно в использовании).

По соглашению многие программы ожидают, что имена файлов будут состоять из основного имени и расширения, разделенных точкой (которая также считается символом). Так, prog.c — это обычно программа на языке C, prog.py — это обычно программа на языке Python, а prog.o — чаще всего объектный файл (выходные данные компилятора). Эти соглашения никак не регламентируются операционной системой, но некоторые компиляторы и другие программы ожидают файлов именно с такими расширениями. Расширения имеют произвольную длину, причем файлы могут иметь по несколько расширений, например, prog.java.Z, что, скорее всего, представляет собой сжатую программу на языке Java.

Для удобства файлы могут группироваться в каталоги. Каталоги хранятся на диске в виде файлов, и с ними можно работать практически так же, как с файлами. Каталоги могут содержать подкаталоги, что приводит к иерархической файловой системе. Корневой каталог называется / и всегда содержит несколько подкаталогов. Символ / используется также для разделения имен каталогов, поэтому имя /usr/ast/x обозначает файл x, расположенный в каталоге ast, который в свою очередь находится в каталоге usr. Некоторые основные каталоги (находящиеся у вершины дерева каталогов) показаны в табл. 4.3.1.

Таблица 4.3.1. Некоторые важные каталоги, существующие в большинстве систем Linux

Каталог	Содержание
bin	Двоичные (исполняемые) программы
dev	Специальные файлы для устройств ввода-вывода
etc	Разные системные файлы
lib	Библиотеки
usr	Каталоги пользователей

Существует два способа задания имени файла в системе Linux (как в оболочке, так и при открытии файла из программы). Первый способ заключается в использовании **абсолютного пути** (absolute path), указывающего, как найти файл от корневого каталога. Также имена путей могут указываться относительно рабочего каталога. Путь, заданный относительно рабочего каталога, называется **относительным путем** (relative path).

Кроме обычных файлов Linux поддерживает также символьные специальные файлы и блочные специальные файлы. Символьные специальные файлы используются для моделирования последовательных устройств ввода-вывода, таких как клавиатуры и принтеры. Блочные специальные файлы (обычно с такими именами, как `/dev/hd1`) могут использоваться для чтения и записи необработанных (raw) дисковых разделов, минуя файловую систему. При этом поиск байта номер *k*, за которым последует чтение, приведет к чтению *k*-го байта из соответствующего дискового раздела, игнорируя i-узел и файловую структуру.

Для безопасного совместного доступа к файлам стандарт POSIX предоставляет гибкий и детальный механизм, позволяющий процессам за одну неделимую операцию блокировать даже единственный байт файла (или целый файл). Механизм блокировки требует от вызывающей стороны указать блокируемый файл, начальный байт и количество байтов. Если операция завершается успешно, то система создает запись в таблице, в которой указывается, что определенные байты файла (например, запись базы данных) заблокированы.

Стандартом определены два типа блокировки: блокировка с монополизацией (exclusive locks) и блокировка без монополизации (shared locks). Если часть файла уже имеет блокировку без монополизации, то повторная попытка установки блокировки без монополизации на это место файла разрешается, но попытка установить блокировку с монополизацией будет отвергнута. Если же какая-либо область файла содержит блокировку с монополизацией, то любые попытки заблокировать любую часть этой области файла будут отвергаться, пока не будет снята блокировка. Для успешной установки блокировки необходимо, чтобы каждый байт в блокируемой области был доступен.

4.3.7. Основные свойства UNIX

Как пользователю, так и системному программисту Unix представляется в виде очень простой системы [6].

Простота и была, по-видимому, одним из важнейших критериев при выборе того или иного решения.

Основные свойства Unix определяются тремя главными составляющими:

- Языком Си, обусловливающим мобильность системы;
- Файловой системой, унифицирующей все средства передачи информации в Unix;
- Командным языком, поднимающим интерфейс пользователя с системой до уровня современных языков программирования.

4.3.8. Мобильность операционных систем

Мобильность, или переносимость, означает возможность и легкость переноса операционной системы на другую аппаратную платформу. Мобильная операционная система обычно разрабатывается с помощью специального языка высокого уровня, предназначенного для создания системного программного обеспечения. Такой язык помимо поддержки высокоуровневых операторов, типов данных и модульных конструкций должен позволять непосредственно использовать аппаратные возможности и особенности процессора. Этот язык системного программирования должен быть

достаточно распространенным и технологичным. Одним из таких языков является язык С. В последние годы язык С++ также стал использоваться для этих целей, поскольку идеи объектно-ориентированного программирования оказались плодотворными не только для прикладного, но и для системного программирования. Большинство современных операционных систем были созданы именно как объектно-ориентированные [7].

Если при разработке операционной системы сразу не следовать принципу мобильности, то в последующем очень трудно обеспечить перенос на другую платформу как самой операционной системы, так и программного обеспечения, созданного для нее.

Одним из аспектов совместимости является способность операционной системы выполнять программы, написанные для других систем или для более ранних версий

данной операционной системы, а также для другой аппаратной платформы. Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой операционной системе. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего транслятора в составе системного программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

Одним из средств обеспечения совместимости программных и пользовательских интерфейсов является соответствие стандартам POSIX. Эти стандарты позволяют создавать программы в стиле UNIX, которые впоследствии могут легко переноситься из одной системы в другую.

4.3.9. Методы инсталляции и настройки ОС

Последовательность действий при установке ОС [8]:

1. Загрузка соответствующей ОС с какого-либо внешнего носителя и запуск программы-инсталлятора.
2. Подготовка накопителя (HDD), на который ОС будет устанавливаться (разметка, форматирование).
3. Перенос с установочного носителя на диск ОС и всего, что ей (и пользователю) необходимо. (На установщик возложена одна очень важная функция - обеспечение контроля зависимостей между пакетами).
4. Обеспечение загрузки установленной системы после рестарта машины.
5. Настройка графического режима работы.

Установка

Третий этап установки - это разворачивание компонентов системы, хранящихся обычно в архивированном и компрессированном виде, и помещение их на смонтированные файловые системы на дисковых разделах. Однако этому предшествует стадия выбора компонентов, подлежащих установке.

Дистрибутивы Linux организованы по пакетному принципу. Пакет – это наименьшая часть, на которую ее можно разделить систему Linux. Пакет может быть добавлен в систему только целиком, и также целиком удален.

В состав дистрибутива пакеты могут быть включены в трех формах:

- в виде исходных текстов (sources - "исходников"), которые перед установкой и использованием подлежат определенным действиям - трансформации в исполняемый код (процесс этот называется сборкой);
- в виде набора правил для получения (из Сети) и сборки готовых к использованию пакетов из исходников - так называемых портов, хотя в Linux такие наборы правил имеют собственные названия в каждом дистрибутиве; при этом сами исходники в состав дистрибутива входить не обязаны;
- в виде заранее собранных, т.н. бинарных (или прекомпилированных) пакетов, которые необходимо только развернуть из архивов и поместить куда нужно.

Обеспечение загрузки

Стандартный способ загрузки ядра — это специальная программа, которая называется системный загрузчик.

Теоретически, можно обойтись и без него, записав в соответствующее место диска (загрузочный сектор) некоторый код, позволяющий обратиться к ядру Linux без посредников. Но на практике в Linux принято два системных загрузчика: традиционный Lilo и GRUB. Оба они не просто системные, а мультисистемные, то есть позволяют загружать не только Linux, но и многие другие ОС, от Windows до любой BSD.

4.3.10. Особенности процессов

Существует очень много версий систем Unix и Linux, и между ними имеются определенные различия. Поэтому здесь основное внимание будет уделено общим свойствам всех версий, а не особенностям какой-либо одной версии. Таким образом, в определенных разделах (особенно в тех, где будет рассматриваться вопрос реализации) может оказаться, что описание не соответствует в равной мере всем версиям [1].

Основными активными сущностями в системе Linux являются процессы. Процессы Linux очень похожи на классические последовательные процессы, которые мы изучали ранее. Каждый процесс выполняет одну программу и изначально получает один поток управления. Иначе говоря, у процесса есть один счетчик команд, который отслеживает следующую исполняемую команду. Linux позволяет процессу создавать дополнительные потоки (после того, как он начинает выполнение).

Linux представляет собой многозадачную систему и несколько независимых процессов могут работать одновременно. Более того, у каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать сотни и даже тысячи процессов. Фактически на большинстве однопользовательских рабочих станций (даже когда пользователь куда-либо отлучился) работают десятки фоновых процессов, называемых демонами (daemons). Они запускаются при загрузке системы из сценария оболочки.

Типичным демоном является **cron**. Он просыпается раз в минуту, проверяя, не нужно ли ему что-то сделать. Если у него есть работа, он ее выполняет, а затем отправляется спать дальше (до следующей проверки). Этот демон позволяет планировать в системе Linux активность на минуты, часы, дни и даже месяцы вперед.

Процессы создаются в операционной системе Linux чрезвычайно просто. Системный вызов **fork** создает точную копию исходного процесса, называемого родительским процессом (parent process). Новый процесс называется дочерним процессом (child process). У родительского и у дочернего процессов есть собственные (приватные) образы памяти. Если родительский процесс впоследствии изменяет какие-либо свои переменные, то эти изменения остаются невидимыми для дочернего процесса (и наоборот).

Открытые файлы используются родительским и дочерним процессами совместно. Это значит, что если какой-либо файл был открыт в родительском процессе до выполнения системного вызова **fork**, он останется открытым в обоих процессах и в дальнейшем. Изменения, произведенные с этим файлом любым из процессов, будут видны другому. Такое поведение является единственно разумным, так как эти изменения будут видны также любому другому процессу, который тоже откроет этот файл.

Тот факт, что образы памяти, переменные, регистры и все остальное у родительского и дочернего процессов идентичны, приводит к небольшому затруднению: как процессам узнать, какой из них должен исполнять родительский код, а какой — дочерний? Секрет в том, что системный вызов **fork** возвращает дочернему процессу число 0, а родительскому — отличный от нуля PID (Process IDentifier — идентификатор процесса) дочернего процесса. Оба процесса обычно проверяют возвращаемое значение и действуют так, как показано в листинге 10.1.

Листинг 10.1. Создание процесса в системе Linux

```
pid = fork( );           /* если fork завершился успешно, pid > 0 в
                            родительском процессе */
if (pid < 0) {
  handle_error();        /* fork потерпел неудачу (например, память
                            или какая-либо таблица переполнена) */
} else if (pid > 0) {
                         /* здесь располагается родительский код */
} else {
                         /* здесь располагается дочерний код */
}
```

Процессы именуются своими PID-идентификаторами. Как уже говорилось, при создании процесса его PID выдается родителю нового процесса. Если дочерний процесс желает узнать свой PID, то он может воспользоваться системным вызовом **getpid**.

Идентификаторы процессов используются различным образом. Например, когда дочерний процесс завершается, его родитель получает PID только что завершившегося дочернего процесса. Это может быть важно, так как у родительского процесса может быть много дочерних процессов. Поскольку у дочерних процессов также могут быть дочерние процессы, то исходный процесс может создать целое дерево детей, внуков, правнуков и более дальних потомков.

Типы процессов

Системные процессы. Системные процессы являются частью ядра и всегда расположены в оперативной памяти [9]. Системные процессы не имеют соответствующих им программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы, таким образом они могут вызывать функции и обращаться к данным, недоступным для остальных процессов. Системными процессами являются: shed (диспетчер свопинга), vhand (диспетчер страничного замещения), bdfflush (диспетчер буферного кэша) и kmadaemon (диспетчер памяти ядра). К системным процессам следует отнести init, являющийся прародителем всех остальных процессов в UNIX. Хотя init не является частью ядра, и его запуск происходит из исполняемого файла (`/etc/init`), его работа жизненно важна для функционирования всей системы в целом.

Демоны. Демоны — это неинтерактивные процессы, которые запускаются обычным образом — путем загрузки в память соответствующих им программ (исполняемых файлов), и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы (но после инициализации ядра,) и обеспечивают работу различных подсистем UNIX: системы терминального доступа, системы печати, системы сетевого доступа и сетевых услуг и т. п. Демоны не связаны ни с одним пользовательским сеансом работы и не могут непосредственно управляться пользователем. Большую часть

времени демоны ожидают пока тот или иной процесс запросит определенную услугу, например, доступ к файловому архиву или печать документа.

Прикладные процессы. К прикладным процессам относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы. С такими процессами вы будете сталкиваться чаще всего. Например, запуск команды **`ls`** породит соответствующий процесс этого типа. Важнейшим пользовательским процессом является основной командный интерпретатор (login shell), который обеспечивает вашу работу в UNIX. Он запускается сразу же после вашей регистрации в системе, а завершение работы login shell приводит к отключению от системы.

Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме, но в любом случае время их жизни (и выполнения) ограничено сеансом работы пользователя. При выходе из системы все пользовательские процессы будут уничтожены. Интерактивные процессы монопольно владеют терминалом, и пока такой процесс не завершит свое выполнение, пользователь не сможет работать другими приложениями. Вы сможете работать с другими приложениями, если в функции интерактивного процесса входит запуск на выполнение других программ. Примером такой задачи является командный интерпретатор shell, который считывает пользовательский ввод и запускает соответствующие задачи.

4.3.11. Сигналы. Обработка сигналов

Сигналы — это способ информирования процесса со стороны ядра о происшествии некоторого события. Смысл термина «сигнал» состоит в том, что сколько бы однотипных событий в системе не произошло, по поводу каждой такой группы событий процессу будет подан ровно один сигнал. То есть, сигнал означает, что определяемое им событие произошло, но не несет информации о том, сколько именно произошло однотипных событий. Сигналы могут инициироваться одними процессами по отношению к другим процессам с помощью специального системного вызова **`kill`**.

По сути, сигналы являются программными прерываниями [1]. Один процесс может послать другому так называемый сигнал (signal). Процессы могут сообщить системе, какие действия следует предпринимать, когда придет входящий сигнал. Варианты такие: проигнорировать сигнал, перехватить его, позволить сигналу убить процесс (действие по умолчанию для большинства сигналов). Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуру обработки сигналов. Когда сигнал прибывает, управление сразу же передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова передается в то место, в котором оно находилось, когда пришел сигнал (это аналогично обработке аппаратных прерываний ввода-вывода). Процесс может посылать сигналы только членам своей группы процессов (process group), состоящей из его прямого родителя (и других предков), братьев и сестер, а также детей (и прочих потомков). Процесс может также послать сигнал сразу всей своей группе за один системный вызов.

Сигналы используются и для других целей. Например, если процесс выполняет вычисления с плавающей точкой и непреднамеренно делит на 0 (делает то, что осуждается математиками), то он получает сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой). Сигналы, требуемые стандартом POSIX, перечислены в табл. 4.3.2. В большинстве систем Linux имеются также дополнительные сигналы, но использующие их программы могут оказаться непереносимыми на другие версии Linux и UNIX.

Таблица 4.3.2. Сигналы, требуемые стандартом POSIX

Сигнал	Причина
SIGABRT	Посылается, чтобы прервать процесс и создать дамп памяти
SIGALRM	Истекло время будильника
SIGFPE	Произошла ошибка при выполнении операции с плавающей точкой (например, деление на 0)
SIGHUP	Сигнал, посылаемый процессу для уведомления о потере соединения с управляющим терминалом пользователя.
SIGILL	Сигнал возникает при попытке выполнить недопустимую, привилегированную или неправильно сформированную инструкцию.
SIGQUIT	Пользователь нажал клавишу, требующую выполнения дампа памяти
SIGKILL	Посылается, чтобы уничтожить процесс (не может игнорироваться или перехватываться)
SIGPIPE	Процесс пишет в канал, из которого никто не читает
SIGSEGV	Процесс обратился к неверному адресу памяти
SIGTERM	Вежливая просьба к процессу завершить свою работу
SIGUSR1	Может быть определен приложением
SIGUSR2	Может быть определен приложением

4.3.12. Неименованные каналы

В системе Linux процессы могут общаться друг с другом с помощью некой формы передачи сообщений. Можно создать канал между двумя процессами, в который один процесс сможет писать поток байтов, а другой процесс сможет его читать. Эти каналы иногда называют трубами (pipes). Синхронизация процессов достигается путем блокирования процесса при попытке прочитать данные из пустого канала. Когда данные появляются в канале, процесс разблокируется [1].

При помощи каналов организуются конвейеры оболочки. Когда оболочка видит строку вроде

```
sort <f | head
```

она создает два процесса, **sort** и **head**, а также устанавливает между ними канал таким образом, что стандартный поток вывода программы **sort** соединяется со стандартным потоком ввода программы **head**. При этом все данные, которые пишет **sort**, попадают напрямую к **head**, для чего не требуется временного файла. Если канал переполняется, то система приостанавливает работу **sort** до тех пор, пока **head** не удалит из него хоть сколько-нибудь данных.

Неименованный канал (или программный канал) представляется в виде области памяти на внешнем запоминающем устройстве, управляемой операционной системой, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы, т.е. это область памяти является разделяемым ресурсом [10].

Для доступа к неименованному каналу система ассоциирует с ним два файловых дескриптора. Один из них предназначен для чтения информации из канала, т.е. с ним можно ассоциировать файл, открытый только на чтение. Другой дескриптор предназначен для записи информации в канал. Соответственно, с ним может быть ассоциирован файл, открытый только на запись.

Организация данных в канале использует стратегию FIFO, т.е. информация, первой записанная в канал, будет и первой прочитанной из канала. Это означает, что для данных файловых дескрипторов недопустимы работы по перемещению файлового

указателя. В отличие от файлов канал не имеет имени. Кроме того, в отличие от файлов неименованный канал существует в системе, пока существуют процессы, его использующие. Предельный размер канала, который может быть выделен процессам, декларируется параметрами настройки операционной системы.

Для создания неименованного канала используется системный вызов **pipe**().

```
#include <unistd.h>
int pipe(int *fd);
```

Аргументом данного системного вызова является массив fd из двух целочисленных элементов. Если системный вызов **pipe**() прорабатывает успешно, то он возвращает код ответа, равный нулю, а массив будет содержать два открытых файловых дескриптора. Соответственно, в fd[0] будет содержаться дескриптор чтения из канала, а в fd[1] — дескриптор записи в канал. После этого с данными файловыми дескрипторами можно использовать всевозможные средства работы с файлами, поддерживающие стратегию FIFO, т.е. любые операции работы с файлами, за исключением тех, которые касаются перемещения файлового указателя.

Неименованные каналы в общем случае предназначены для организации взаимодействия родственных процессов, осуществляющегося за счет передачи по наследству ассоциированных с каналом файловых дескрипторов. Но иногда встречаются вырожденные случаи использования неименованного канала в рамках одного процесса.

4.3.13. Именованные каналы

Файловая система ОС Unix поддерживает некоторую совокупность файлов различных типов. Файловая система рассматривает каталоги как файлы специального типа каталог, обычные файлы, с которым мы имеем дело в нашей повседневной жизни, — как регулярные файлы, устройства, с которыми работает система, — как специальные файлы устройств. Точно так же файловая система ОС Unix поддерживает специальные файлы, которые называются FIFO-файлами (или именованными каналами). Файлы этого типа очень схожи с обыкновенными файлами (в них можно писать и из них можно читать информацию), за исключением того факта, что они организованы по стратегии FIFO (т.е. невозможны операции, связанные с перемещением файлового указателя) [11].

Таким образом, файлы FIFO могут использоваться для организации взаимодействия процессов, при этом в отличие от неименованных каналов эти файлы могут существовать независимо от процессов, взаимодействующих через них. Эти файлы хранятся на внешних запоминающих устройствах, поэтому возможно открыть этот файл, записать в него информацию, а через любой промежуток времени (в течение которого допустимы перезагрузки системы) прочитать записанную информацию.

Для создания файлов FIFO в различных реализациях используются разные системные вызовы, одним из которых может являться **mkfifo**().

```
int mkfifo(char *pathname, mode_t mode);
```

Первым параметром является имя создаваемого файла, а второй параметр отвечает за флаги владения, права доступа и т.п.

После создания именованного канала любой процесс может установить с ним связь посредством системного вызова **open**(). При этом действуют следующие правила:

- если процесс открывает FIFO-файл для чтения, он блокируется до тех пор, пока какой-либо процесс не откроет тот же канал на запись;
- если процесс открывает FIFO-файл на запись, он будет заблокирован до тех пор, пока какой-либо процесс не откроет тот же канал на чтение;
- процесс может избежать такого блокирования, указав в вызове **open**() специальный флаг (в разных версиях ОС он может иметь разное символьное

обозначение — O_NONBLOCK или O_NDELAY). В этом случае в ситуациях, описанных выше, вызов **open**() сразу же вернет управление процессу.

Правила работы с именованными каналами, в частности, особенности операций чтения-записи, полностью аналогичны неименованным каналам.

4.3.14. Архитектура ОС UNIX. Принципы организации и структура ОС. Ядро и основные компоненты ОС.

Unix - это семейство операционных систем (ОС), обладающих сходной архитектурой и интерфесом с пользователем. Unix как явление зародилось в начале 70-х годов и развивается до сих пор. Основные современные варианты UNIX: Linux, BSD (FreeBSD, NetBSD, OpenBSD), AIX, HPUX, Solaris, SCO. Важнейшие стандарты, обеспечивающие целостность семейства UNIX:

- POSIX - Portable Operating System Interface
- ANSI C (c89 и c99)

Классическая архитектура UNIX двухуровневая [9]:

- Ядро — управляет ресурсами компьютера и предлагает программам базовый набор услуг (системные вызовы).
- Системные программы (управление сетью, терминалами, печатью), прикладные программы (редакторы, утилиты, компиляторы и т.д.).

Операционная система UNIX обладает классическим монолитным ядром, в котором можно выделить следующие основные части: файловая подсистема, управление процессами и драйверы устройств (рис. 4.3.2) [3]

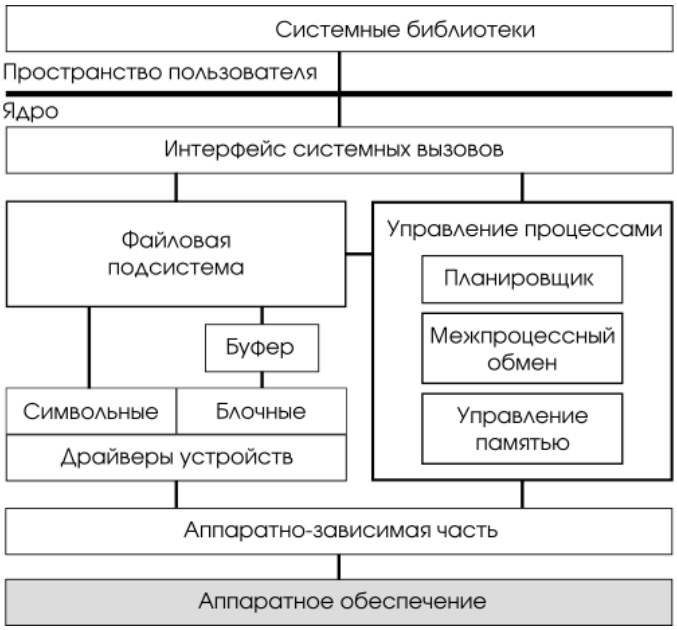

Рис. 4.3.2. Ядро операционной системы UNIX

Драйверы устройств делятся на символьные и блочные по типу внешнего устройства. Для каждого из устройств определен набор возможных операций (открытие, чтение и т.д.). Блочные устройства кэшируются с помощью специального внутреннего механизма управления буферами.

Функции ядра:

- инициализация системы - загрузка и запуск ОС
- управление процессами и потоками
- управление памятью - отображение адресного пространства на физическую память, совместное использование памяти процессами

- управление файлами - реализует понятие файловой системы, дерева каталогов и файлов
- обмен данными между процессами, выполняющимися внутри одного компьютера, в разных узлах сетей передачи данных, а также между процессами и драйверами внешних устройств
- программный интерфейс (API) - обеспечивает доступ к возможностям ядра со стороны процессов пользователя через системные вызовы, оформленных в виде библиотеки функций на Си.

Ядро изолирует программы пользователя от аппаратуры. Все части системы, не считая небольшой части ядра, полностью независимы от архитектуры компьютера и написаны на Си. Системные вызовы - это уровень, скрывающий особенности конкретного механизма выполнения на уровне аппаратуры от программ пользователя. Для программиста, системный вызов - это функция (определенная на Си), которую он вызывает в своей программе. Все низкоуровневые операции осуществляются через системные вызовы.

Подсистемы ядра:

- **Файловая подсистема**. Обеспечивает унифицированный доступ к файлам:
  - контроль прав доступа к файлу;
  - чтение/запись файла;
  - размещение и удаление файла;
  - перенаправление запросов к периферийным устройствам, соответствующим модулям подсистемы ввода/вывода.
- **Подсистема управления процессами**. Запущенная на выполнение программа порождает один или несколько процессов (задач). Подсистема контролирует:
  - создание и удаление процессов
  - распределение системных ресурсов (памяти, вычислительных ресурсов) между процессами
  - синхронизация процессов
  - межпроцессное взаимодействие Специальная задача ядра планировщик процессов разрешает конфликты процессов в конкуренции за ресурсы.
- **Подсистема ввода/вывода**. Выполняет запросы файловой системы и подсистемы управления процессами для доступа к периферийным устройствам (дискам, лентам, терминалам). Обеспечивает буферизацию данных и взаимодействует с драйверами устройств.

Оболочка (shell) - это программа, которая позволяет вам связываться с операционной системой. Она считывает команды, которые вы вводите, и интерпретирует их как запросы на выполнение других программ, на доступ к файлу или обеспечение вывода. Оболочка также поддерживает мощный язык программирования, который обеспечивает условное выполнение и управление потоками данных.

4.3.15. iOS

iOS (до 24 июня 2010 года — iPhone OS) — мобильная операционная система для смартфонов, электронных планшетов, носимых проигрывателей, разрабатываемая и выпускаемая американской компанией Apple. Выпущена в 2007 году; первоначально — для iPhone и iPod touch, позже — для таких устройств, как iPad. В 2014 году появилась поддержка автомобильных мультимедийных систем Apple CarPlay. В отличие от Android (Google), выпускается только для устройств, производимых фирмой Apple [12].

В iOS используется ядро XNU, основанное на микроядре Mach и содержащее программный код, разработанный компанией Apple, а также код из ОС NeXTSTEP и FreeBSD. Ядро iOS почти идентично ядру настольной операционной системы Apple macOS

(ранее называвшейся OS X). Начиная с самой первой версии iOS работает только на планшетных компьютерах и смартфонах с процессорами архитектуры ARM.

Пользовательский интерфейс iOS основан на концепции прямого взаимодействия с использованием жестов «мультитач». Элементы управления интерфейсом состоят из ползунков, переключателей и кнопок.

iOS разработана на основе операционной системы OS X (позднее переименованной в macOS) и использует тот же набор основных компонентов Darwin, совместимый со стандартом POSIX.

Слои абстракции iOS:

- Core OS;
- Core Services;
- Media Layer;
- Cocoa Touch.

4.3.18. MacOS

Mac OS (Macintosh Operating System) — семейство проприетарных операционных систем с графическим интерфейсом. Apple хотела, чтобы Макинтош представлялся как компьютер «для всех». Сам термин «Mac OS» в действительности не существовал до тех пор, пока не был официально использован в середине 1990-х годов.

Компания Apple была так же первая, кто придумал и использовал компьютерную мышь. Которая стала очень популярным устройством.

По данным компании Net Applications, в июле 2022 года рыночная доля составляла: 4.86%

История ОС Mac OS

Mac OS вышла в свет в 1984 году вместе с первым персональным компьютером Macintosh от компании Apple. Идеи, воплощенные в первой версии системы Mac OS, ее авторы почерпнули у фирмы Xerox.

Пользователи управляли своим компьютером не только вводимыми с клавиатуры командами и инструкциями, но и с помощью нового в те времена устройства, названного мышью.

В исследовательском центре Xerox PARC в то время уже существовал компьютер с графической операционной системой, что тогда было настоящим прорывом в эволюционном развитии операционных систем. Но они использовали ее только для собственных нужд и не планировали коммерческого применения.

Соединив уже имеющиеся наработки и собственные идеи, программисты компании Apple создали Mac OS, первую доступную для всех графическую операционную систему. В ней уже тогда был использован всем нам привычный оконный интерфейс, папки с файлами, и впервые был применен манипулятор, названный компьютерной мышью, способный передвигать курсор по всей области экрана. Такая концепция вполне соответствовала главной идее самой компании Apple, предлагавшей создать компьютер доступный для всех, как по цене, так и в техническом плане.

Первая версия Mac OS занимала всего 216 кб дискового пространства и работала даже при обычном копировании с одного компьютера на другой. Но такой продукт был совершенно не защищен от подделки, поэтому для того, чтоб сохранить свои доходы разработчики все дальнейшее время посвятили не только ее техническому усовершенствованию, расширению функциональности и стабильности, но и защите.

Теперь в **Mac OS X** используется ядро Mach, стандартные сервисы BSD и все основные возможности операционной системы Unix. Это дало возможность в много раз повысить ее функциональность, защищенность и стабильность. Вытесняющая многозадачность,

которая используется в Mac OS X, позволяет работать нескольким процессам сразу, но при этом не мешать друг другу, а при сбое в работе одного из них не допускать сбоя всей системы и прерывания работы других процессов.

После выхода версии Mac OS X 10.0 было выпущено еще шесть ее модификаций, каждая из которых носит название животного из семейства кошачьих (Cheetah (Гепард), Puma, Jaguar, Tiger, Leopard, Snow Leopard))

На данный момент Mac OS X имеет собственный красивый, не перегруженный спецэффектами и приятный для глаз интерфейс Aqua. Она проста в использовании и дружелюбна. В ней используется среда программирования Core Foundation. К тому же Mac OS X позволяет использовать программное обеспечение, написанное на таких языках программирования, как Си, C++, Objective-C, Ruby и Java. Немаловажным достоинством Mac OS X является ее безопасность при работе в интернете, она неплохо защищена от интернет-атак, да и количество вирусов способных ее поразить на сегодняшний день ничтожно мало.

Особенности для программиста

1. Unix-подобная ОС:
    - macOS основана на ядре Unix, что делает ее близкой к другим Unix-подобным системам, таким как Linux. Это обеспечивает программистам доступ к широкому спектру инструментов командной строки, а также возможность разработки с использованием стандартных для Unix технологий.
    - macOS основана на Unix, что обеспечивает программистам доступ к Unix-подобным инструментам. В то время как Windows является преимущественно осью, ориентированной на предприятия.
    - macOS поставляется с Xcode и поддерживает язык программирования Swift. Windows, с другой стороны, обычно используется для разработки на C#, C++, и Java, с использованием инструментов, таких как Visual Studio.
2. Xcode и Swift:
    - Xcode — это интегрированная среда разработки (IDE) от Apple, предназначенная для создания приложений под macOS, iOS, watchOS и tvOS. Она включает в себя множество инструментов для разработки, отладки и профилирования приложений. Swift, современный язык программирования, был разработан Apple и является основным языком для создания приложений под macOS.

      Visual Studio - популярная IDE, широко используемая для разработки под Windows. Однако, существуют и другие IDE для обеих платформ.
3. App Store:
    - macOS имеет свой собственный App Store, который упрощает установку и обновление приложений. Это может быть удобно для конечных пользователей, но для программистов также существует возможность распространения приложений вне App Store.
    - Homebrew — это популярная система управления пакетами для macOS, позволяющая устанавливать, обновлять и управлять программными пакетами из командной строки.

      Windows предоставляет Microsoft Store, а также установку программ из сторонних источников. Обновление системы и программ происходит с помощью Windows Update.
4. Ограниченная совместимость с аппаратным обеспечением:
    - macOS ограничена аппаратным обеспечением, производимым Apple. Это означает, что разработчики могут ориентироваться на конкретные конфигурации Mac, что облегчает тестирование и оптимизацию приложений.

- Windows существует на широком спектре устройств от различных производителей, что означает более широкую совместимость с разными конфигурациями.

5. Docker и Виртуализация:
   - macOS поддерживает использование Docker и виртуализации с помощью инструментов, таких как Parallels или VirtualBox, что облегчает тестирование и развертывание приложений в различных средах.

Плюсы и Минусы

Главной причиной небольшого процента людей, которые выбрали для работы компьютеры от компании Apple, является в первую очередь цена, а во вторую очередь закрытость MacOS. Политика руководства компании направлена на то, чтоб операционная система MacOS могла устанавливаться только на компьютеры их собственного производства, поэтому те, кто решил насладиться всеми преимуществами Mac OS, просто обязаны купить себе Macintosh.

Модельный ряд компьютеров Macintosh, предложенный в магазинах, довольно бедный, каждое направление представлено лишь двумя-тремя экземплярами. При том все компьютеры выпускаются только в готовом виде, и тем, кто привык сам собирать себе компьютер, этот вариант совсем не подойдет. Но с другой стороны, придя в магазин, вам не придется долго думать, какой из Macintosh стоит выбрать, при этом качество каждого из них будет на самом высоком уровне

При этом ОС Mac OS была создана именно для компьютеров Macintosh, что позволяет вам использовать возможности оборудования на 100 процентов, а не переплачивать деньги за новинки.

К тому же в комплект с Mac OS входит набор действительно полезных программ, позволяющих организовать весь рабочий процесс современного человека. Также плюсом является бесплатное сервисное обслуживание в любом фирменном магазине компании Apple.

Еще одной неприятной проблемой является закрытость ОС Mac OS, что в первую очередь сказывается на недостатке программного обеспечения для нее от сторонних разработчиков. До сих пор еще не существует некоторых важных программных продуктов, написанных под Macintosh, да и любителям игр разгуляться не получится, поскольку игры разрабатываются в первую очередь для Windows, а потом уже для Mac OS, к тому некоторых игрушек вы вообще не найдете.

Но время не стоит на месте, и появляются организации, которые занимаются разработкой программных продуктов под MacOS, а известные разработчики программного обеспечения заинтересованы в том, чтоб их продукт работал на компьютерах Macintosh.

Споры, что лучше можно продолжать до бесконечности, но, если вы спросите у тех, кто решился и приобрел себе компьютер Macintosh, согласен ли он его поменять на другой, скорей всего вы получите отрицательный ответ. Те, кто работает на Macintosh – любят свои компьютеры. Объяснить это можно тем, что руководство компании Apple создает свои продукты в первую очередь для людей. Главной их стратегией является красота и удобство. К тому же все их разработки идут в ногу со временем, и даже немного его опережают.

4.3.16. ОС Android

Android — относительно новая операционная система, сконструированная для работы на мобильных устройствах. Она основана на ядре Linux — для самой операционной системы Android в ядро Linux введено всего лишь несколько новых понятий и используется большинство уже знакомых вам средств Linux (процессы, идентификаторы

пользователей, виртуальная память, файловые системы, планирование и т. д.), иногда весьма отличными от их первоначального предназначения способами [1].

Android выросла в одну из наиболее распространенных операционных систем для смартфонов. Ее популярность пришла на волне взрывного распространения смартфонов, и она находится в свободном доступе для производителей мобильных устройств, что позволяет использовать ее в их продукции. Она также является платформой с открытым кодом, что позволяет подстраиваться под широкое многообразие устройств. Она приобрела популярность не только для устройств, рассчитанных на покупателей, где в выгодном свете представлялось наличие экосистемы из приложений сторонних разработчиков (вроде приложений для планшетных компьютеров, телевизоров, игровых систем и медиаплееров), но все чаще находит применение в качестве встроенной операционной системы для специализированных устройств, нуждающихся в графическом интерфейсе пользователя (GUI), таких как VOIP-телефоны, смарт-часы, приборные панели автомобилей, медицинские устройства и бытовые приборы.

Значительная часть операционной системы Android написана на языке программирования высокого уровня Java. Ядро и большое количество библиотек низкого уровня написаны на C и C++. Но существенная часть системы написана на Java, и весь API приложений, за весьма небольшим исключением, также написан и издан на Java. Те части Android, которые написаны на Java, имеют ярко выраженную тенденцию следования объектно-ориентированной модели, чему, собственно, способствует сам язык.

В Android открытый исходный код сочетается со сторонними приложениями с закрытым исходным кодом. Часть Android с открытым исходным кодом называется Android Open Source Project (AOSP) и является полностью открытой с возможностью повсеместного использования и изменения.

Документ определения совместимости (Compatibility Definition Document (CDD)) дает описание способов поведения Android, позволяющих добиться совместимости с приложениями сторонних разработчиков. В самом документе описывается, что нужно Android-устройству для поддержки совместимости. Но без некоторых методов принуждения к такой совместимости его требования могли бы часто игнорироваться. Для соблюдения совместимости необходим какой-то дополнительный механизм.

В Android эта проблема была решена разрешением надстраивать дополнительные частные службы над платформой с открытым кодом с предоставлением служб (как правило, облачных), которые не могла реализовывать сама платформа. Поскольку такие службы были частными, они могли ограничивать круг включаемых в них устройств, требуя, таким образом, от этих устройств CDD-совместимости.

Google создала Android, чтобы получить возможность поддержки широкого круга частных облачных служб наряду с широким набором служб, представленных самой компанией Google: почтовой службы Gmail, службы синхронизации календаря и контактов, службы обмена сообщениями между облаком и устройством и многих других, часть из которых была видима пользователю, а часть — нет. Когда же дело касается предложения совместимых приложений, то наиболее важной службой является Google Play.

Google Play — это онлайн-магазин компании Google для Android-приложений. Как правило, когда разработчики создают Android-приложения, они выставляют их в Google Play. Поскольку Google Play (или любой другой магазин приложений) является каналом, по которому приложения доставляются на Android-устройство, эта частная служба отвечает за то, что приложения будут работать на тех устройствах, которым она их доставляет.

В Google Play используются два основных механизма обеспечения совместимости. Первый и наиболее важный заключается в требовании того, что любое устройство,

поставляемое с этим магазином, должно быть совместимым Android-устройством, соответствующим CDD. Тем самым гарантируется основная линия поведения всех устройств. Кроме того, магазин Google Play должен знать обо всех свойствах устройства, требуемых приложением (например, о наличии GPS для осуществления навигации по карте), чтобы приложение не было доступно на тех устройствах, у которых это свойство отсутствует.

Архитектура Android

Android является надстройкой над стандартным ядром Linux, имеющей всего несколько существенных расширений самого ядра, которые будут рассмотрены позже. А вот в пространстве пользователя его реализация сильно отличается от традиционных распространяемых версий Linux и использует множество свойств Linux совершенно иными способами.

В традиционной Linux-системе первым Android-процессом в пользовательском пространстве является init, который является корневым для всех других процессов. Но демонов init-процесс Android запускает по-другому, концентрируясь на низкоуровневых деталях (управлении файловыми системами и доступом к оборудованию), а не на высокоуровневых пользовательских возможностях, таких как планирование заданий с определенным сроком выполнения (cron jobs). У Android также имеется дополнительный уровень процессов, которые запускают среду языка Java под названием Dalvik.

Эта среда отвечает за выполнение всех частей системы, реализованных на языке Java. Присущая Android основная структура процессов показана на рис. 4.3.3. Первым показан процесс init, который дает начало ряду низкоуровневых процессов-демонов. Одним из них является процесс zygote, корневой для высокоуровневых процессов языка Java.

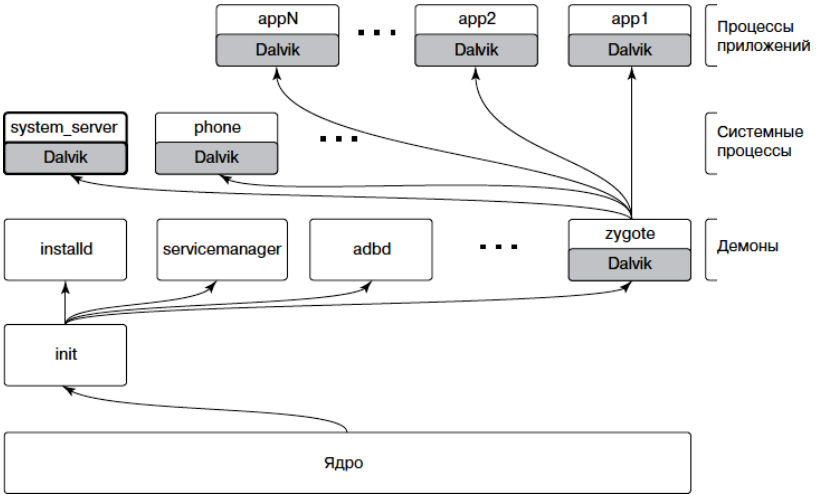

Рис. 4.3.3. Иерархия процессов Android

Android-процесс init не запускает оболочку традиционным способом, потому что обычное Android-устройство не имеет локальной консоли для доступа к оболочке. Вместо этого демон-процесс adbd прислушивается к удаленным подключениям (например, по USB), запрашивающим доступ к оболочке, по необходимости ответвляя для них процессы оболочки.

Поскольку основная часть Android написана на языке Java, центральными для системы являются демон zygote и запущенные им процессы. Первый процесс, всегда запускаемый zygote, называется system_server и содержит все основные службы операционной системы. Основными частями являются диспетчер электропитания, диспетчер пакетов, оконный диспетчер и диспетчер активностей.

По мере необходимости из zygote будут создаваться другие процессы. Некоторые из них станут постоянными процессами, являющимися частью основной операционной системы,

например, телефонный стек в процессе телефона, который должен всегда оставаться в работе. В ходе работы системы по мере необходимости будут создаваться и останавливаться дополнительные процессы приложений.

Взаимодействие приложений с операционной системой осуществляется за счет вызовов предоставляемых ею библиотек, совокупность и является средой Android (Android framework). Некоторые из библиотек могут работать в рамках данного процесса, но многим понадобится межпроцессный обмен данными с другими процессами. Зачастую этот обмен ведется со службами в процессе system_server.

Типовая конструкция API-функций среды Android, взаимодействующей с системными службами, в данном случае с диспетчером пакетов, показана на рис. 4.3.4. Диспетчер пакетов предоставляет API-функции среды для приложений, чтобы они могли сделать вызов в своем локальном процессе, в данном случае вызов класса PackageManager.

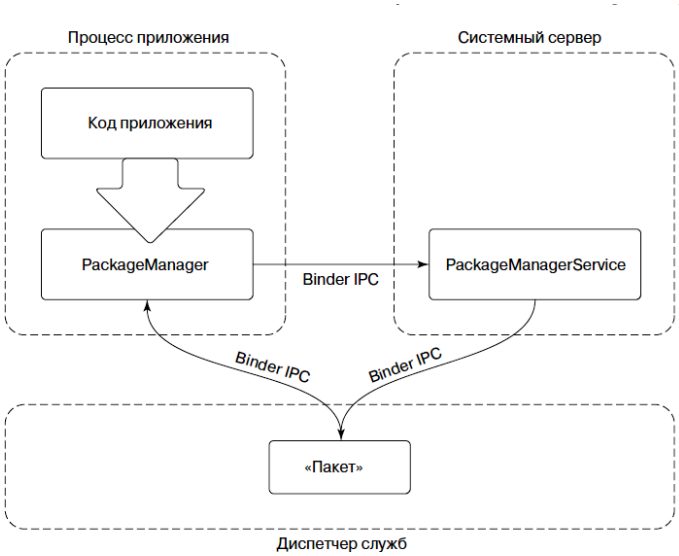

Рис. 4.3.4. Публикация системных служб и взаимодействие с ними

Внутри среды этот класс должен получить подключение к соответствующей службе в system_server. Для выполнения этой задачи в ходе начальной загрузки system_server публикует каждую службу под четко определенным именем в диспетчере служб (service manager), демоне-процессе, запускаемом процессом init. PackageManager в процессе приложения извлекает подключение из service manager в его системную службу, используя это же имя.

Как только PackageManager подключится к системной службе, он сможет осуществлять адресуемые ей вызовы. Большинство вызовов со стороны приложений к PackageManager реализуются как обмен данными между процессами с использованием Android-механизма Binder IPC, в данном случае осуществляя вызовы к PackageManagerService в system_server. Реализация PackageManagerService разрешает конфликты среди клиентских приложений и поддерживает состояние, которое будет необходимо нескольким приложениям.

Список использованных источников

1. [ТА] Таненбаум, Э. Современные операционные системы. / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с.

2[acm18] Введение в UNIX
https://acm.bsu.by/wiki/Unix2018/Введение_в_UNIX

3. [heap] Архитектура UNIX. Особенности архитектуры UNIX.
http://heap.altlinux.org/modules/unix_base_admin.dralex/ch01s02.html

4. [stud 56] Пользователи системы, Атрибуты пользователя
https://studfile.net/preview/1424056/page:11/

5.[wm] Атрибуты пользователя
https://wm-help.net/lib/b/book/173895509/73

6. [stud3650-2] Основные свойства Unix
https://studfile.net/preview/3650/page:2/

7[stud106]. Принцип мобильности при построении ос
https://studfile.net/preview/9509106/page:4/

8. Тема 2.1 UNIX-подобные и другие POSIX-совместимые операционные системы
https://www.polessu.by/sites/default/files/sites/default/files/02per/032.pdf

9 [kv] Общая характеристика ОС семейства Unix. Основные компоненты (структура) Unix-системы. Виды программ (процессов) в Unix.
https://kvckr.me/mag/sp/2.html

10 [stud90] Неименованные каналы
https://studfile.net/preview/7477290/page:60/

11[stud61] Именованные каналы
https://studfile.net/preview/7477290/page:61/

12. iOS
https://ru.wikipedia.org/wiki/IOS