Оглавление

[1. Что такое ОС? Взгляды "снизу" (управление ресурсами), "сверху" (абстракции), "со стороны" (эксплуатация, жизненный цикл). 3](#_Toc472680724)

[2. История развития ОС. 5](#_Toc472680725)

[3. Отечественные ОС. 11](#_Toc472680726)

[4. Свободные лицензии (BSD, GPL разных версий и др.). Свободный код не означает общедоступный или бесплатный. Совместимость, возможный переток кода, примеры. 14](#_Toc472680727)

[5. Коммерческие лицензии (OEM, BOX, аренда, подписка). Примеры. 16](#_Toc472680728)

[6. Классификация ОС. Не менее пяти критериев, пояснить выбор и содержание критериев. 18](#_Toc472680729)

[7. Процессы (абстракция). Состояния (жизненный цикл) процессов, создание, завершение. Реализация (таблица процессов). Модель многозадачности, оценка утилизации ресурсов процессора, накладные расходы. Процесс как контейнер ресурсов. 21](#_Toc472680730)

[8. Потоки (абстракция). Различные подходы к реализации, сравнение. Возможные проблемы при использовании потоков. 25](#_Toc472680731)

[9. Планирование при многозадачности, задачи. Критерии оценки эффективности, примеры. Вытесняющая и не вытесняющая многозадачности. Приоритеты, оптимальность, справедливость. Планирование потоков и процессов 30](#_Toc472680732)

[10. Оперативная память. Адресация. Адресное пространство (абстракция). Подкачка. Виртуальная память, реализация (таблица). Алгоритмы заполнения и вытеснения страниц. 34](#_Toc472680733)

[11. Носители информации. Файлы и каталоги (абстракции). Структура файловой системы. Развитие атрибутов, прав доступа. Основные операции. Текущий каталог, переменные окружения. 43](#_Toc472680734)

[12. Промежуточные уровни абстракции: менеджер логических томов и виртуальная файловая система. Специализация файловых систем. Отказоустойчивость файловых систем (дублирование метаинформации, журналирование, распределение копий данных). Направления развития. 50](#_Toc472680735)

[13. Виртуализация. Типы виртуализации, применение. Облако как продукт виртуализации 53](#_Toc472680736)

[14. Пользователи и группы (абстракции). Модели управления доступом. Пользователь как контейнер прав, наследование прав. Идентификация, аутентификация, авторизация. Алгоритмы/методики аутентификации (в т.ч. многофакторная) 58](#_Toc472680737)

[15. Системы централизованного управления настройками ОС. Цели, примеры (AD GPO, Puppet/Chef), архитектура. Управление обновлениями (патч-менеджмент) 62](#_Toc472680738)

[16. Основные технологии защиты в ОС. DEP, ASLR, UAC/sudo 65](#_Toc472680739)

[17. Начальный этап загрузки - BIOS/UEFI, MBR/GPT 67](#_Toc472680740)

Оглавление 2 (дополнительные вопросы)

[1. Ситуации (сценарии) когда эффективнее использование процессов, когда - потоков 70](#_Toc472683031)

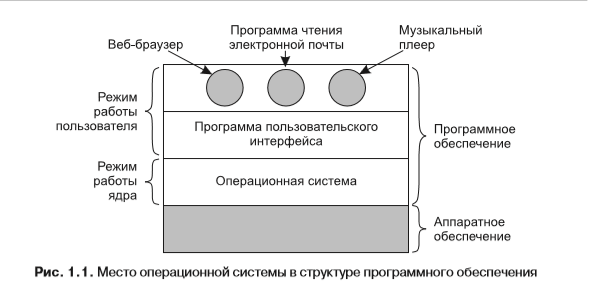
[4. Классические задачи взаимодействия процессов: обедающие философы, читатели и писатели, спящий парикмахер 72](#_Toc472683032)

[5.Гарвардская и фоннеймановские модели памяти 77](#_Toc472683033)

[6. RAID-массивы 78](#_Toc472683034)

1. Что такое ОС? Взгляды "снизу" (управление ресурсами), "сверху" (абстракции), "со стороны" (эксплуатация, жизненный цикл).

Компьютеры оснащены специальным уровнем программного обеспечения, который называется операционной системой, в чью задачу входит управление пользовательскими программами, а также всеми ранее упомянутыми ресурсами (одного или нескольких процессоров, оперативной памяти, дисков, принтера, клавиатуры, мыши, дисплея, сетевых интерфейсов и других разнообразных устройств ввода-вывода). Программы, с которыми взаимодействуют пользователи, обычно называемые оболочкой, когда они основаны на применении текста, и графическим пользовательским интерфейсом (Graphical User Interface (GUI)), когда в них используются значки, фактически не являются частью операционной системы, хотя задействуют эту систему в своей работе. Большинство компьютеров имеют два режима работы: режим ядра и режим пользователя. Операционная система — наиболее фундаментальная часть программного обеспечения, работающая в режиме ядра (этот режим называют еще режимом супервизора). В этом режиме она имеет полный доступ ко всему аппаратному обеспечению и может задействовать любую инструкцию, которую машина в состоянии выполнить. Вся остальная часть программного обеспечения работает в режиме пользователя, в котором доступно лишь подмножество инструкций машины.



Все программы, работающие в режиме ядра, безусловно, являются частью операционной системы, но некоторые программы, работающие вне этого режима, возможно, также являются ее частью или, по крайней мере, имеют с ней тесную связь.

Понятно, почему операционные системы живут так долго, — их очень трудно создавать, и, написав одну такую систему, владелец не испытывает желания ее выбросить и приступить к созданию новой. Поэтому операционные системы развиваются в течение долгого периода времени. Семейство Windows 95/98/Me по своей сути представляло одну операционную систему, а семейство Windows NT/2000/XP/Vista/ Windows 7 — другую.

Дать точное определение операционной системы довольно трудно. Можно сказать, что это программное обеспечение, которое работает в режиме ядра, но и это утверждение не всегда будет соответствовать истинному положению вещей. Отчасти проблема здесь в том, что операционные системы осуществляют две значительно отличающиеся друг от друга функции: предоставляют прикладным программистам (и прикладным программам, естественно) вполне понятный абстрактный набор ресурсов взамен неупорядоченного набора аппаратного обеспечения и управляют этими ресурсами.

Сверху:  
Абстракции: пример – жесткий диск->драйвера->файлы.

Хорошая абстракция превращает практически неподъемную задачу в две, решить которые вполне по силам. Первая из этих задач состоит в определении и реализации абстракций, а вторая — в использовании этих абстракций для решения текущей проблемы. Задача операционной системы заключается в создании хорошей абстракции, а затем в реализации абстрактных объектов, создаваемых в рамках этой абстракции, и управлении ими.

Снизу:  
Управление ресурсами (менеджер ресурсов).

Современные компьютеры состоят из процессоров, памяти, таймеров, дисков, мышей, сетевых интерфейсов, принтеров и широкого спектра других устройств. Сторонники взгляда снизу вверх считают, что задача операционной системы заключается в обеспечении упорядоченного и управляемого распределения процессоров, памяти и устройств ввода-вывода между различными программами, претендующими на их использование.

Современные операционные системы допускают одновременную работу нескольких программ. Сторонники этого взгляда на операционную систему считают, что ее первичной задачей является отслеживание того, какой программой какой ресурс используется, чтобы удовлетворять запросы на использование ресурсов, нести ответственность за их использование и принимать решения по конфликтующим запросам от различных программ и пользователей.

Управление ресурсами включает в себя мультиплексирование (распределение) ресурсов двумя различными способами: во времени и в пространстве. Когда ресурс разделяется во времени, различные программы или пользователи используют его по очереди: сначала ресурс получают в пользование одни, потом другие и т. д. К примеру, располагая лишь одним центральным процессором и несколькими программами, стремящимися на нем выполняться, операционная система сначала выделяет центральный процессор одной программе, затем, после того как она уже достаточно поработала, центральный процессор получает в свое распоряжение другая программа, затем еще одна программа, и, наконец, его опять получает в свое распоряжение первая программа. Определение того, как именно ресурс будет разделяться во времени — кто будет следующим потребителем и как долго, — это задача операционной системы. Другим примером мультиплексирования во времени может послужить совместное использование принтера. Когда в очереди для распечатки на одном принтере находятся несколько заданий на печать, нужно принять решение, какое из них будет выполнено следующим. Другим видом разделения ресурсов является пространственное разделение. Вместо поочередной работы каждый клиент получает какую-то часть разделяемого ресурса. Например, оперативная память обычно делится среди нескольких работающих программ, так что все они одновременно могут постоянно находиться в памяти (например, используя центральный процессор по очереди). При условии, что памяти достаточно для хранения более чем одной программы, эффективнее разместить в памяти сразу несколько программ, чем выделять всю память одной программе, особенно если ей нужна лишь небольшая часть от общего пространства. Разумеется, при этом возникают проблемы равной доступности, обеспечения безопасности и т. д., и их должна решать операционная система. Другим ресурсом с разделяемым пространством является жесткий диск. На многих системах на одном и том же диске могут одновременно храниться файлы, принадлежащие многим пользователям. Распределение дискового пространства и отслеживание того, кто какие дисковые блоки использует, — это типичная задача операционной системы по управлению ресурсами.

1. История развития ОС.

Так как операционные системы появились и развивались в процессе конструирования компьютеров, эти события исторически тесно связаны.

**Первое поколение (1945–1955): электронные лампы.**

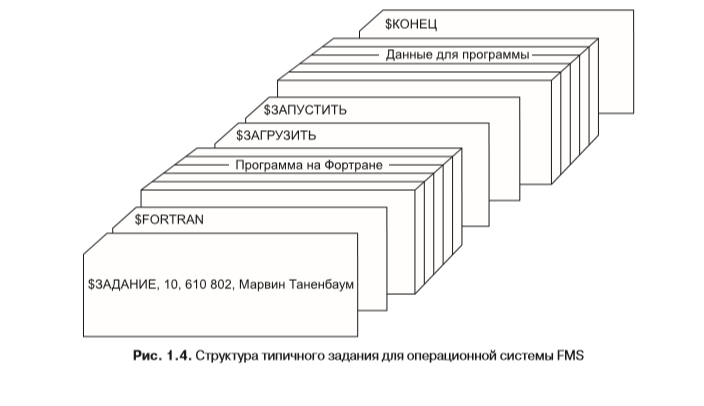
На заре компьютерной эры каждую машину проектировала, создавала, программировала, эксплуатировала и обслуживала одна и та же группа людей (как правило, инженеров). Все программирование велось исключительно на машинном языке или, и того хуже, за счет сборки электрических схем, а для управления основными функциями машины приходилось подключать к коммутационным панелям тысячи проводов. О языках программирования (даже об ассемблере) тогда еще ничего не было известно. Об операционных системах вообще никто ничего не слышал. Когда в начале 1950-х годов появились перфокарты, положение несколько улучшилось. Появилась возможность вместо использования коммутационных панелей записывать программы на картах и считывать с них, но в остальном процедура работы не претерпела изменений.

**Второе поколение (1955–1965): транзисторы и системы пакетной обработки.**

В середине 1950-х годов изобретение и применение транзисторов радикально изменило всю картину. Компьютеры стали достаточно надежными, появилась высокая вероятность того, что машины будут работать довольно долго, выполняя при этом полезные функции. Впервые сложилось четкое разделение между проектировщиками, сборщиками, операторами, программистами и обслуживающим персоналом.

Если учесть высокую стоимость оборудования, неудивительно, что люди довольно скоро занялись поиском способа повышения эффективности использования машинного времени. Общепринятым решением стала система пакетной обработки. Первоначально замысел состоял в том, чтобы собрать полный поднос заданий (колод перфокарт) в комнате входных данных и затем переписать их на магнитную ленту, используя небольшой и (относительно) недорогой компьютер, например IBM 1401, который был очень хорош для считывания карт, копирования лент и печати выходных данных, но не подходил для числовых вычислений.





Большие компьютеры второго поколения использовались главным образом для научных и технических вычислений, таких как решение дифференциальных уравнений в частных производных, часто встречающихся в физике и инженерных задачах. В основном программы для них составлялись на языке Фортран и ассемблере, а типичными операционными системами были FMS (Fortran Monitor System) и IBSYS (операционная система, созданная корпорацией IBM для компьютера IBM 7094).

**Третье поколение (1965–1980): интегральные схемы и многозадачность.**

К началу 1960-х годов большинство производителей компьютеров имели два различных, не совместимых друг с другом семейства. С одной стороны, это были огромные научные компьютеры с пословной обработкой данных типа IBM 7094, которые использовались для промышленного уровня числовых расчетов в науке и технике, с другой — коммерческие компьютеры с посимвольной обработкой данных, такие как IBM 1401, широко используемые банками и страховыми компаниями для задач сортировки и распечатки данных. Развитие и поддержка двух совершенно разных семейств была для производителей весьма обременительным делом.

Фирма IBM попыталась решить эти проблемы разом, выпустив серию машин IBM System/360. Семейство компьютеров IBM/360 стало первой основной серией, использующей малые интегральные схемы, дававшие преимущество в цене и качестве по сравнению с машинами второго поколения, собранными на отдельных транзисторах.

По замыслу его создателей, все программное обеспечение, включая операционную систему OS/360, должно было одинаково хорошо работать на всех моделях компьютеров. Но ни IBM, ни кому-либо еще так и не удалось создать программное обеспечение, удовлетворяющее всем этим противоречивым требованиям. В результате появилась громоздкая и чрезвычайно сложная операционная система, примерно на два или три порядка превышающая по объему FMS. Она состояла из миллионов строк, написанных на ассемблере тысячами программистов, содержала тысячи и тысячи ошибок, что повлекло за собой непрерывный поток новых версий, в которых предпринимались попытки исправления этих ошибок. В каждой новой версии устранялась только часть ошибок, вместо них появлялись новые, так что общее их количество, скорее всего, оставалось постоянным. Несмотря на свой огромный объем и имеющиеся недостатки, OS/360 и подобные ей операционные системы третьего поколения, созданные другими производителями компьютеров, неплохо отвечали запросам большинства клиентов. Они даже сделали популярными несколько ключевых технических приемов, отсутствовавших в операционных системах второго поколения. Самым важным достижением явилась многозадачность.

Решение заключалось в разбиении памяти на несколько частей, называемых разделами, в каждом из которых выполнялось отдельное задание (рис. 1.5). Пока одно задание ожидало завершения работы устройства ввода-вывода, другое могло использовать центральный процессор. Если в оперативной памяти содержалось достаточное количество заданий, центральный процессор мог быть загружен почти на все 100 % времени. Множество одновременно хранящихся в памяти заданий требовало наличия специального оборудования для защиты каждого задания от возможного незаконного присваивания областей памяти и нанесения вреда со стороны других заданий. Для этой цели компьютеры 360-й серии и другие системы третьего поколения были оборудованы специальными аппаратными средствами.

Другим важным плюсом операционных систем третьего поколения стала способность считывать задание с перфокарт на диск по мере того, как их приносили в машинный зал. При окончании выполнения каждого текущего задания операционная система могла загружать новое задание с диска в освободившийся раздел памяти и запускать это задание. Этот технический прием называется подкачкой данных, или спулингом.

Хотя операционные системы третьего поколения неплохо справлялись с большинством научных вычислений и крупных коммерческих задач по обработке данных, но по своей сути они были все еще разновидностью систем пакетной обработки. В системах третьего поколения промежуток времени между передачей задания и возвращением результатов часто составлял несколько часов, так что единственная поставленная не в том месте запятая могла стать причиной сбоя при компиляции, и получалось, что программист полдня тратил впустую.

Желание сократить время ожидания ответа привело к разработке режима разделения времени — варианту многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Первая универсальная система с режимом разделения времени CTSS (Compatible Time Sharing System) была разработана в Массачусетском технологическом институте (M.I.T.) на специально переделанном компьютере IBM 7094 (Corbato et al., 1962). Однако режим разделения времени не стал действительно популярным до тех пор, пока на машинах третьего поколения не получили широкого распространения необходимые технические средства защиты.

После создания успешной системы CTSS Массачусетский технологический институт, исследовательские лаборатории Bell Labs и корпорация General Electric (главный на то время изготовитель компьютеров) решили начать разработку универсальной общей компьютерной системы — машины, которая должна была поддерживать одновременную работу сотен пользователей в режиме разделения времени. За основу была взята система распределения электроэнергии. Когда вам нужна электроэнергия, вы просто вставляете штепсель в розетку и получаете столько энергии, сколько вам нужно. Проектировщики этой системы, известной как MULTICS (MULTiplexed Information and Computing Service — мультиплексная информационная и вычислительная служба), представляли себе одну огромную вычислительную машину, воспользоваться услугами которой мог любой проживающий в окрестностях Бостона человек.

К концу XX века идея создания такого общего компьютера уже выдохлась, но она может возродиться в виде облачных вычислений, когда относительно небольшие компьютеры (включая смартфоны, планшеты и им подобные устройства) подключены к серверам, принадлежащим огромным удаленным центрам обработки данных, где и производятся все вычисления, а локальный компьютер используется просто для обслуживания пользовательского интерфейса.

Несмотря на коммерческую неудачу, система MULTICS оказала существенное влияние на последующие операционные системы (особенно на UNIX и ее производные, на FreeBSD, Linux, IOS и Android).

Еще одной важной разработкой времен третьего поколения были мини-компьютеры, невероятный взлет популярности которых начался с выпуска корпорацией DEC машины PDP-1. Компьютеры PDP-1 обладали оперативной памятью, состоящей всего лишь из 4 К 18-битовых слов, но стоили всего 120 тыс. долларов за одну машину (это меньше 5 % от цены IBM 7094) и поэтому расхватывались как горячие пирожки. Некоторые виды нечисловой работы они выполняли так же быстро, как и машины IBM 7094, в результате чего родилась новая отрасль производства. За этой машиной вскоре последовала целая серия компьютеров PDP других моделей (в отличие от семейства IBM, полностью несовместимых), и как кульминация появилась модель PDP-11. Кен Томпсон (Ken Thompson), один из ведущих специалистов Bell Labs, работавший над проектом MULTICS, чуть позже нашел мини-компьютер PDP-7, которым никто не пользовался, и решил написать упрощенную однопользовательскую версию системы MULTICS. Эта работа позже переросла в операционную систему **UNIX®**, ставшую популярной в академических кругах, правительственных учреждениях и во многих компаниях.

Чтобы появилась возможность писать программы, работающие в любой UNIX-системе, Институт инженеров по электротехнике и электронике (IEEE) разработал стандарт системы UNIX, названный POSIX, который в настоящее время поддерживается большинством версий UNIX. Стандарт POSIX определяет минимальный интерфейс системных вызовов, который должны поддерживать совместимые с ним системы UNIX. Фактически на данный момент POSIX-интерфейс поддерживается также рядом других операционных систем.

Желание получить свободно распространяемую версию MINIX (в отличие от образовательной) привело к тому, что финский студент Линус Торвальдс (Linus Torvalds) создал систему Linux.

**Четвертое поколение (с 1980 года по наши дни): персональные компьютеры.**

Следующий период эволюции операционных систем связан с появлением БИС — больших интегральных схем (LSI, Large Scale Integration) — кремниевых микросхем, содержащих тысячи транзисторов на одном квадратном сантиметре. С точки зрения архитектуры персональные компьютеры (первоначально называемые микрокомпьютерами) были во многом похожи на мини-компьютеры класса PDP-11, но, конечно же, отличались по цене(дешевле).

В 1974 году, когда корпорация Intel выпустила Intel 8080 — первый универсальный 8-разрядный центральный процессор, — для него потребовалась операционная система, с помощью которой можно было бы протестировать новинку. Корпорация Intel привлекла к разработкам и написанию нужной операционной системы одного из своих консультантов Гэри Килдэлла. Килдэлл создал дисковую операционную систему, названную CP/M (Control Program for Microcomputers — управляющая программа для микрокомпьютеров). Когда Килдэлл заявил о своих правах на CP/M, корпорация Intel удовлетворила его просьбу, поскольку не думала, что у микрокомпьютеров с диском есть будущее. Позже Килдэлл создал компанию Digital Research для дальнейшего развития и продажи CP/M.

В начале 1980-х корпорация IBM разработала IBM PC (Personal Computer — персональный компьютер)1 и начала искать для него программное обеспечение. Сотрудники IBM связались с Биллом Гейтсом, чтобы получить лицензию на право использования его интерпретатора языка Бейсик. Корпорация IBM снова обратилась к Гейтсу с просьбой обеспечить ее операционной системой. После повторного обращения Гейтс выяснил, что у местного изготовителя компьютеров, Seattle Computer Products, есть подходящая операционная система DOS (Disk Operating System — дисковая операционная система). Он направился в эту компанию с предложением выкупить DOS (предположительно за $50 000), которое компания Seattle Computer Products с готовностью приняла. Затем Гейтс создал пакет программ DOS/BASIC, и пакет был куплен IBM. Когда корпорация IBM захотела внести в операционную систему ряд усовершенствований, Билл Гейтс пригласил для этой работы Тима Патерсона (Tim Paterson), человека, написавшего DOS и ставшего первым служащим Microsoft. Видоизмененная система была переименована в MS-DOS (MicroSoft Disk Operating System) и быстро заняла доминирующее положение на рынке IBM PC.

CP/M, MS-DOS и другие операционные системы для первых микрокомпьютеров полностью основывались на командах, вводимых пользователем с клавиатуры. Со временем благодаря исследованиям, проведенным в 1960-е годы Дагом Энгельбартом (Doug Engelbart) в научно-исследовательском институте Стэнфорда (Stanford Research Institute), ситуация изменилась. Энгельбарт изобрел графический интерфейс пользователя (GUI, Graphical User Interface) вкупе с окнами, значками, системами меню и мышью.

Однажды Стив Джобс увидел GUI и сразу понял уровень заложенного в него потенциала. Джобс приступил к созданию компьютера Apple, оснащенного графическим пользовательским интерфейсом. Этот проект привел к созданию компьютера Lisa, который оказался слишком дорогим и не имел коммерческого успеха. Вторая попытка Джобса, компьютер Apple Macintosh, имел огромный успех потому, что обладал более дружественным пользовательским интерфейсом, предназначенным для пользователей, не разбиравшихся в компьютерах. В 1999 создана **Mac OS X,** которая является операционной системой, построенной на основе UNIX, хотя и с весьма своеобразным интерфейсом.

Когда корпорация Microsoft решила создать преемника MS-DOS, она была под большим впечатлением от успеха Macintosh. В результате появилась основанная на применении графического интерфейса пользователя система под названием Windows, первоначально являвшаяся надстройкой над MS-DOS. На протяжении примерно 10 лет, с 1985 по 1995 год, Windows была просто графической оболочкой, работавшей поверх MS-DOS. Однако в 1995 году была выпущена самостоятельная версия Windows — Windows 95. Она непосредственно выполняла большинство функций операционной системы, используя входящую в ее состав систему MS-DOS только для загрузки, а также для выполнения старых программ, разработанных для MS-DOS. В 1998 году была выпущена слегка модифицированная версия этой системы, получившая название Windows 98. Тем не менее обе эти системы, и Windows 95 и Windows 98, все еще содержали изрядное количество кода, написанного на ассемблере для 16-разрядных процессоров Intel. Другой операционной системой Microsoft была Windows NT (NT означает New Technology — новая технология), которая на определенном уровне совместима с Windows 95. Однако она была написана заново и представляла собой полноценную 32-разрядную систему. Ведущим разработчиком Windows NT был Дэвид Катлер. Пятая версия Windows NT была в начале 1999 года переименована в Windows 2000. Она предназначалась для замены обеих версий — Windows 98 и Windows NT 4.0. Но полностью этим планам также не суждено было сбыться, поэтому Microsoft выпустила еще одну версию Windows 98 под названием Windows Me (Millennium edition — выпуск тысячелетия). В 2001 году была выпущена слегка обновленная версия Windows 2000, названная Windows XP. Эта версия выпускалась намного дольше, по существу заменяя все предыдущие версии Windows. После Windows 2000 Microsoft разбила семейство Windows на клиентскую и серверную линейки. Клиентская линейка базировалась на версии XP и ее последователях, а серверная включала Windows Server 2003 и Windows 2008. Чуть позже появилась и третья линейка, предназначенная для мира встроенных операционных систем.

Затем в январе 2007 года Microsoft выпустила окончательную версию преемника Windows XP под названием Vista. У нее был новый графический интерфейс, усовершенствованная система безопасности и множество новых или обновленных пользовательских программ. Microsoft надеялась, что она полностью заменит Windows XP, но этого так и не произошло. Вместо этого было получено большое количество критических отзывов и статей в прессе, главным образом из-за высоких системных требований, ограничительных условий лицензирования и поддержки технических средств защиты авторских прав (технологии, затрудняющей пользователям копирование защищенных материалов). С появлением Windows 7, новой и менее требовательной к ресурсам операционной системы, многие решили вообще пропустить Vista. В Windows 7 не было представлено слишком много новых свойств, но она была относительно небольшой по объему и довольно стабильной. Менее чем за три недели Windows 7 получила большую долю рынка, чем Vista за семь месяцев. В 2012 году Microsoft выпустила ее преемника — Windows 8, операционную систему с совершенно новым внешним видом, предназначенным для сенсорных экранов.

Другим основным конкурентом в мире персональных компьютеров является операционная система UNIX (и различные производные от этой системы). UNIX имеет более сильные позиции на сетевых и промышленных серверах, также она находит все более широкое распространение и на настольных компьютерах, ноутбуках, планшетных компьютерах и смартфонах. На компьютерах с процессором Pentium популярной альтернативой Windows для студентов и постоянно растущего числа корпоративных пользователей становится операционная система Linux.

Операционная система FreeBSD также является популярной производной от UNIX, порожденной проектом BSD в Беркли. Все современные компьютеры Macintosh работают на модифицированной версии FreeBSD (OS X). UNIX также является стандартом на рабочих станциях, оснащенных высокопроизводительными RISC-процессорами. Ее производные нашли широкое применение на мобильных устройствах, которые работают под управлением iOS 7 или Android.

В середине 1980-х годов начало развиваться интересное явление — рост сетей персональных компьютеров, работающих под управлением сетевых операционных систем и распределенных операционных систем (Tanenbaum and Van Steen, 2007). В сетевых операционных системах пользователи знают о существовании множества компьютеров и могут войти в систему удаленной машины и скопировать файлы с одной машины на другую. На каждой машине работает своя локальная операционная система и имеется собственный локальный пользователь (или пользователи).

**Пятое поколение (с 1990 года по наши дни): мобильные компьютеры**

Первый настоящий мобильный телефон появился в 1946 году, и тогда он весил около 40 кг.

Первый по-настоящему переносной телефон появился в 1970-х годах и при весе приблизительно 1 кг был воспринят весьма позитивно. Первый настоящий смартфон появился только в середине 1990-х годов, когда Nokia выпустила свой N9000, представлявший собой комбинацию из двух отдельных устройств: телефона и КПК. В 1997 году в компании Ericsson для ее изделия GS88 «Penelope» был придуман термин «смартфон».

На момент написания этих строк доминирующей была операционная система Google Android, а на втором месте находилась Apple iOS.

В первое десятилетие после своего появления большинство смартфонов работало под управлением Symbian OS. Эту операционную систему выбрали такие популярные бренды, как Samsung, Sony Ericsson, Motorola и Nokia.

Для производителей телефонов Android обладала тем преимуществом, что имела открытый исходный код и была доступна по разрешительной лицензии. Кроме того, у этой операционной системы имеется огромное сообщество разработчиков, создающих приложения в основном на общеизвестном языке программирования Java.

1. Отечественные ОС.

При анализе истории развития области ИТ следует иметь в виду особые условия, в которых развивались эти разработки как в СССР (России), так и в США, начиная с 1950-х гг. – "холодная война" и "железный занавес". Вследствие этого, все эти разработки, как в области аппаратуры, так и в области программного обеспечения, были строго засекречены. Такая ситуация приводила к тому, что аналогичные идеи подчас возникали и реализовывались по обе стороны "железного занавеса" примерно в одно и то же время, при почти полном отсутствии информации о работах друг друга. Отечественные разработчики, почти ничего не зная об аналогичных работах американских коллег, создавали свои оригинальные системы, в том числе – ОС. Например, идея многопоточности (multi-threading) была реализована в ОС "Эльбрус" еще в конце 1970-х гг., а в популярных зарубежных ОС (UNIX, Solaris, Windows NT) многопоточность появилась только в конце 1980-х – начале 1990-х гг. К сожалению, имело место и существенное отставание советских и российских ИТ-специалистов от американцев – прежде всего, в области разработки элементной базы и технологии производства компьютеров, а также в области графических пользовательских интерфейсов (GUI).

Среди передовых оригинальных отечественных разработок в области компьютерной аппаратуры и ОС 1960-х – 1970-х гг. следует выделить прежде всего ЭВМ БЭСМ-6, ее операционные системы: ОС ДИСПАК, ОС ДИАПАК, ОС ИПМ и ее системное и прикладное программное обеспечение. Разработчиком БЭСМ-6, ОС ДИСПАК и ОС ДИАПАК был Институт точной механики и вычислительной техники АН СССР под руководством академика Сергея Алексеевича Лебедева, основателя всей нашей отечественной вычислительной техники. Разработчик ОС ИПМ – Институт прикладной математики АН СССР. ЭВМ БЭСМ-6 и ее программное обеспечение следует признать уникальными. В их развитии участвовали многие академические и университетские коллективы не только СССР, но и зарубежных стран - достаточно вспомнить такие системы, как АЛГОЛ-ГДР - реализацию расширения Алгола-60 с развитыми математическими библиотеками, выполненную нашими коллегами из Германии, а также реализацию Паскаля для БЭСМ-6, разработанную специалистами из Польской Академии наук. Операционные системы для БЭСМ-6 поддерживали пакетный (с учетом приоритетов и ресурсов заданий) и диалоговый режимы взаимодействия с компьютером, страничную организацию виртуальной памяти, работу с внешними устройствами и телекоммуникационными каналами, работу в локальных сетях. К каждой БЭСМ-6 были подключены десятки терминалов, работавших под управлением диалоговых систем ДИМОН, ДЖИН и др. (это при объеме оперативной памяти БЭСМ-6 всего в 32 страницы по 4096 байтов и быстродействии до 1 млн. операций в секунду). Работу БЭСМ-6 и ее ОС отличала высокая надежность. Руководитель разработки ОС ДИСПАК – В.Ф. Тюрин.

Другой передовой отечественной разработкой 1970-х – 1980-х гг. была разработка многопроцессорных вычислительных комплексов (МВК) "Эльбрус-1" и "Эльбрус-2". В разработке программного обеспечения системы "Эльбрус" автор курса со своей командой принимал активное участие в течение 15 лет. Идейным вдохновителем проекта "Эльбрус" стал сам С.А. Лебедев, затем им руководили академик Всеволод Сергеевич Бурцев, а после него – чл.-корр. АН СССР Борис Арташесович Бабаян. Следует признать, что у "Эльбруса" были зарубежные прототипы и задолго до его появления были написаны академические зарубежные работы, заложившие научные основы подобных компьютерных архитектур. Однако разработчикам "Эльбруса" и его операционной системы удалось предложить и реализовать целый ряд собственных оригинальных идей и методов. Основными принципами "Эльбруса", как и его предшественников, являлись: теговая архитектура (каждое слово памяти, кроме данных, содержало тег – код типа данных, хранящихся в этом слове, по которому аппаратура контролировала правильность выполнения операции), динамизм и аппаратная поддержка типичных (подчас весьма сложных) последовательностей действий, используемых при реализации языков высокого уровня - например, вход в процедуру по указателю на нее, с установкой дисплей-регистров, ссылающихся на доступные процедуре области локальных данных. ОС "Эльбрус" поддерживала создание процессов и операции над ними, аналогичные тем, которые впоследствии в зарубежных разработках были названы многопоточностью (multi-threading); была реализована математическая (виртуальная) память с поддержкой страничного распределения виртуальной памяти (на диске) и сегментного распределения физической (оперативной) памяти. Динамизм выражался в том, что отсутствовала статическая линковка; все программы и модули загружались в память только динамически, при первом вызове. Также динамически, при первом запросе, по прерыванию, выделялся каждый массив математической памяти. Подобные принципы были для своего времени передовыми, использование тегов значительно повысило надежность. Однако, с современной точки зрения, идеологию "Эльбруса", по-видимому, нельзя считать гибкой и эффективной, так как все аппаратные операции и соответствующие действия ОС были реализованы в общем виде, и практически отсутствовала какая-либо возможность оптимизаций, например, для более быстрого вызова процедуры в случае отсутствия необходимости обращения к ее аргументам, для быстрого доступа к статической области памяти и т.д.

Были и другие интересные отечественные разработки новых архитектур компьютеров и их операционных систем, прежде всего - оригинальные специализированные компьютеры для различных применений и их операционные системы (в основном, по своему классу и назначению, они были системами реального времени).

В начале 1970-х годов в развитии отечественной вычислительной техники и ее системного программного обеспечения начался новый этап. Правительство СССР приняло беспрецедентное решение о создании, в качестве основной на достаточно долгий период времени отечественной серии - Единой Системы ЭВМ (ЕС ЭВМ) - путем копирования американских компьютеров серии IBM 360. Соответственно, все базовое системное программное обеспечение, в том числе и ОС, также было адаптировано к использованию в СССР (либо использовалось в оригинальном виде – с сообщениями на английском языке и т.д.). Это решение вызвало большие проблемы с финансированием у разработчиков отечественных архитектур компьютеров. Это также вызвало большие сложности у пользователей и разработчиков программного обеспечения, так как далеко не все хорошо владели английским языком. Появились, например, системы-обертки, обеспечивающие русскоязычный интерфейс: с их помощью все задания для ЕС писались с использованием русскоязычной мнемоники, затем конвертировались в англоязычный Job Control Language (язык управления заданиями IBM 360), а все сообщения, выдаваемые в качестве результатов, переводились на русский язык. Это было интересным подходом, однако не прижилось. Документация по IBM 360 постепенно была переведена на русский язык, появилась русскоязычная справочная и учебная литература по ЕС ЭВМ. К сожалению, отечественные аналоги аппаратуры IBM 360 – машины серии ЕС ЭВМ – оказались гораздо менее надежными, чем их прототипы. В течение еще нескольких лет было принято еще одно правительственное решение – об аналогичном копировании американских миникомпьютеров серий PDP 10 и PDP 11, под общим названием "Система Мини-ЭВМ" (СМ ЭВМ). Были выпущены компьютеры этой серии СМ-1, СМ-2, СМ-3 и СМ-4. Были и другие аналогичные работы по копированию зарубежных архитектур компьютеров и выпуске на этой основе отечественных аналогов. Копирование машин IBM 360 и PDP, с одной стороны, дало возможность советским программистам освоить новые развитые операционные системы, языки программирования, библиотеки программ, с другой – отбросило нашу отчественную вычислительную технику еще дальше назад.

Разумеется, история отечественных ОС на этом не закончилась. Например, в настоящее время ведется разработка отечественной свободно распространяемой операционной системы на базе Linux. Среди отечественных программистов многие являются специалистами весьма высокого уровня по операционным системам.

**Не очень российская ROSA Linux**Вот, например, ROSA Linux (читается не как «роза», а как «роса», которая по утрам на траве). Вряд ли ее можно считать полностью российской, потому что это на самом деле дальнейшие ветви развития зарубежных сборок Mandriva и Red Hat. Версия для домашних пользователей называется Rosa Fresh; последний билд вышел в августе 2016 года.  
Преимуществом этой бесплатной сборки по сравнению с той же Ubuntu является возможность выбора графического интерфейса (KDE или Gnome), наличие драйверов «из коробки» для большого количества устройств, включая проприетарные (скажем, видеокарты NVIDIA), и предустановленного ПО — типа Skype, Java, Flash, Steam, собственного всеядного медиаплеера и т.п., а также удобных средств для создания точек восстановления.

**Astra Linux**Astra Linux — еще одна российская сборка Linux (изначально создана на базе Debian), разработанная для силовиков и спецслужб. Она отличается высочайшим уровнем защиты и сертифицирована для работы со сведениями, содержащими государственную тайну. Для пущего патриотизма все релизы названы в честь городов-героев России.  
Сейчас актуальны «Орел» — версия для повседневных офисных, простите, конторских задач и «Смоленск» для работы со сведениями категории «Совершенно секретно». Готовится к выходу «Новороссийск» — мобильная версия ОС для смартфонов и планшетов с ARM-процессорами.  
С технической точки зрения «Астра» отличается от всех других Linux-систем собственной запатентованной системой разграничения доступа, а также имеет ряд других функций защиты данных — например, при удалении файла он удаляется полностью, и место, которое занимал, заполняется случайными маскирующими последовательностями данных (в других ОС по умолчанию меняется лишь запись в FAT, и для того, чтобы удаленный файл нельзя было прочитать посекторным чтением накопителя, используют специальные утилиты).

**ПО для спецслужб**«Заря» — еще одна сборка Linux специального назначения (на базе Red Hat), она используется исключительно в российской армии и существует в виде нескольких сборок — для рабочих станций, для ЦОД, для специализированных компьютеризированных комплексов и т.п.  
Также существует МСВС — «Мобильная система Вооруженных Сил» и GosLinux — ОС для Федеральной службы судебных приставов (тоже на базе Red Hat). По последней есть статистика: она установлена на 660 серверов и 16 тыс. рабочих станций, при этом стоимость копии в пересчете на один компьютер оказалась равна 1500 рублей. По мере установки на большее число ПК к концу 2016 года (тогда она будет установлена на половине компьютеров ФССП) средняя стоимость снизится до 800 рублей. В любом случае это в несколько раз дешевле лицензий на Windows и MS Office.

**«Эльбрус»**«Эльбрус» — операционная система для компьютеров с российскими процессорами с аналогичным названием. Поскольку процессоры эти, хоть и совместимы с x86, имеют собственную уникальную архитектуру, решили разработать специальную ОС — опять же, на ядре Linux — которая учитывает особенности ЦП и максимально эффективно использует их преимущества.

**Фантом ОС**   
— операционная система, разрабатываемая российской компанией Digital Zone с 2010 годаОС Фантом — одна из немногих ОС, не опирающихся на классические концепции Unix-подобных систем (в отличие от их концепции «Всё есть файл», Фантом базируется на принципе «Всё есть объект»). ОС под названием «Фантом», разработана с нуля.  
Одна из ключевых особенностей «Фантома» — это персистентность, что означает, что приложения работают без остановки и даже «не знают» о том, что компьютер был выключен или перезагружен — работа продолжается ровно с того же момента. Это чем-то похоже на режим «гибернации» в других системах (когда содержимое памяти записывается на диск в виде файлов и затем загружается), но гарантированно работает без сбоев драйверов и программ, и все происходит автоматически. Даже если внезапно обесточить компьютер, данные не пропадут и после повторного включения все будет так же, как за несколько секунд до выключения.  
Проблема «Фантома» только одна: под него нужно писать (или портировать из Unix-систем) прикладное ПО, а тут возникает проблема курицы и яйца: пока не будет хоть какого-то проникновения ОС, никто не захочет писать под нее программы, а пока нет программ — не будет проникновения.

1. Свободные лицензии (BSD, GPL разных версий и др.). Свободный код не означает общедоступный или бесплатный. Совместимость, возможный переток кода, примеры.

Свободная лицензия (англ. free license) — такой лицензионный договор (ранее — «авторский договор»), условия которого содержат разрешения пользователю от правообладателя на конкретный перечень способов использования его произведения, которые дают ему четыре важнейшие свободы

Свободное ПО (free software) — это ПО, распространяемое на условиях свободной лицензии (либо находящееся в общественном достоянии), а свободная лицензия — это та, которая даёт пользователю право:

* Использовать программу
* Изменять её исходный код
* Распространять программу
* Распространять изменённые версии

Свободное и открытое ПО — это совсем не то же самое, что бесплатное. Свободное и открытое ПО — это не то же самое, что некоммерческое. Практически все сколько-нибудь крупные свободные проекты (Linux, gcc, glibc, Gnome, FreeBSD, clang и многие другие) являются коммерческими.

Двойное лицензирование — это (если не указано противного) две лицензии, соединённые союзом «или» (то же самое относится к тройному лицензированию и т. д.). Т. е. пользователь может выбрать, на условиях какой из этих лицензий он будет пользоваться материалом.

**Лицензия BSD** (англ. BSD license, Berkeley Software Distribution license — Программная лицензия университета Беркли) — это лицензионное соглашение, впервые применённое для распространения UNIX-подобных операционных систем BSD.

Существуют две основные версии лицензии BSD, которые необходимо различать: «оригинальная» и так называемая «модифицированная» (вторую в англоязычной литературе часто называют New BSD License).

Эти лицензии были подвергнуты ряду изменений, породив множество лицензий, обобщённо именуемых «лицензии типа BSD». «В настоящее время лицензии типа BSD являются одними из самых популярных лицензий для свободного программного обеспечения и используются для многих программ (помимо BSD-версий UNIX, для которых лицензия BSD была изначально создана).»

Лицензия BSD допускает проприетарное коммерческое использование ПО. Для ПО, выпущенного под этой лицензией, допускается встраивание в проприетарные коммерческие продукты. Работы, основанные на таком ПО, даже могут распространяться под проприетарными лицензиями (но всё же обязаны соответствовать требованиям лицензии). Наиболее заметные примеры таких программ — использование сетевого кода BSD в продуктах корпорации Microsoft, а также использование многих компонентов FreeBSD в операционной системе Mac OS X. Плюс недавнее использование ядра FreeBSD для создания PlayStation 4.

Можно применять к распространяемому продукту одновременно лицензию BSD и какую-то другую. Например, так было с самыми ранними версиями самой BSD, которая включала проприетарные материалы из AT&T.

Главная особенность BSD заключается в том, что она не только позволяет использовать продукт в сторонних разработках, но в отличие от GPL разрешает в дальнейшем сделать его закрытым.

**GNU General Public License**

Сокращенное наименование — GNU GPL или просто GPL. Согласно исследованию, проведенному компанией Black Duck Software, 70% всех открытых проектов распространяются под лицензией семейства GPL.

Эта лицензия предоставляет пользователям следующее:

* — право свободно запускать программу с любой целью;
* — право свободно изучать исходный текст программы;
* — право свободно модифицировать и улучшать программу;
* — право свободно распространять её копии.

При этом важно, что ни один пользователь, получивший подобные права, не должен каким-то образом ограничивать аналогичные права других пользователей. То есть если кто-то решил модифицировать программу, то распространять измененный вариант он должен на тех же условиях, на которых получил прототип. Произведение, получив свободу один раз, не может быть лишено ее впоследствии.

GNU Lesser GPL v2.1: Данная версия лицензии датируется 1999 годом и содержит одно огромное отличие от обычной лицензии GNU GPL: предназначенная для библиотек, лицензия позволяет использовать их в проприетарном программном обеспечении. Например, библиотеки GNU C распространяются под лицензией GNU Lesser GPL v2.1, для того, чтобы сторонние разработчики могли использовать их в своем ПО, свободном или коммерческом.

1. Коммерческие лицензии (OEM, BOX, аренда, подписка). Примеры.



Коробочная версия включает в себя лицензионное соглашение, сертификат подлинности, дистрибутив с программным продуктом, а для более ранних версий продуктов — также регистрационную карточку и документацию в печатном виде.[4] Подтверждением лицензионных прав пользователя является Сертификат подлинности (СОА), наклеенный на коробку.

В случае OEM-версии (Original Equipment Manufacturer — англ. производитель нового оборудования) покупатель приобретает оборудование с предустановленными программным обеспечением.[5] Приобретаемое программное обеспечение жёстко привязано к оборудованию и может эксплуатироваться только на нём. OEM-версии программного обеспечения распространяются только среди поставщиков компьютерного оборудования и могут быть проданы только ими и только вместе с оборудованием. Необходимым подтверждением лицензионных прав пользователя является сертификат подлинности, наклеенный на корпус ПК.

Ограничения OEM-лицензий

* - OEM-версия может использоваться только на том ПК, на котором она была первоначально установлена. Это главное ограничение, отличающее OEM-версии от версий, закупленных по другим способам лицензирования.
* - Если клиент приобретает новый компьютер, он должен приобрести для него новую копию OEM-продукта. Перенести на этот компьютер OEM-лицензию со старого ПК нельзя, даже если старый компьютер более не используется.
* - Если пользователь хочет передать или продать старый компьютер с установленной на нем OEM-версией ПО другому пользователю, то вместе с компьютером передается и все части OEM- лицензии.

Немного от майкрософт

Open Value Subscription – это идеально подходящая для малых и средних предприятий программа лицензирования, которая позволяет взять в аренду лицензии и каждый год вносить платеж только за те программы, которые нужны компании в данный момент. Вы можете как увеличивать количество лицензий, так и отказываться от тех, которые стали не нужны бизнесу. Главное преимущество и отличие от других программ лицензирования – это возможность уменьшать нужное количество лицензий, что идеально подойдет для предприятий, которые не могут на данном этапе строить долгосрочные планы касательно нужного ПО.

Особенности лицензирования OVS

При оформлении соглашения все компьютеры компании стандартизируются, т.е. оплачиваются лицензии на базовые продукты (хотя бы на одну категорию – ОС, клиентские лицензии или приложения) для всех ПК предприятия. Количество необходимых базовых лицензий регулируется каждый год, и зависит от количества ПК. Т.е. если компания приобрела дополнительные ПК, в соглашение добавляются лицензии, если ПК стало меньше – количество лицензий уменьшается (минимальное количество устройств на компанию должно быть не меньше 5). Подписка на дополнительные, не базовые продукты, оформляется в любое время, а оплачивается в месяц заказа.

А это отдельно, но похоже, мы имеем дело с одним и тем же

Подписка на лицензирование программного обеспечения предусматривает внесение ежемесячных или ежегодных платежей. Эта схема удобна компаниям, которые покупают более 10 лицензий. Она позволяет пользователям за минимальные начальные затраты получить практически все основные преимущества использования данного продукта.

1. Классификация ОС. Не менее пяти критериев, пояснить выбор и содержание критериев.

**(из наших лекций)**

ОС принято классифицировать по следующим критериям:

1. Однопользовательские/многопользовательские
2. Однозадачные/многозадачные  
   допольнительно многозадачные классифицируются следующим образом:

* вытесняющие
* невытесняюшие
* реального времени

1. сертифицированная/несесртифицированная
2. поддерживает виртуальную память/не поддерживает
3. кроссплатформенные/нет

Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей.  
Следует заметить, что не всякая многозадачная система является многопользовательской, и не всякая однопользовательская ОС является однозадачной.

Однозадачные операционные системы в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.  
Многозадачные операционные системы, кроме вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

Основным различием вытесняющего и не вытесняющего вариантов многозадачности является степень централизации механизма планирования процессов. В случае не вытесняющей многозадачности механизм планирования процессов целиком сосредоточен в операционной системе, а в случае вытесняющей многозадачности он распределен между системой и прикладными программами.  
При не вытесняющей многозадачности активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению процесс.  
При вытесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается операционной системой, а не самим активным процессом.

**Или с Интуита**

Все многообразие существующих (и ныне не использующихся) ОС можно классифицировать по множеству различных признаков. Остановимся на основных классификационных признаках.

**1.По назначению** ОС делятся на универсальные и специализированные.   
Специализированные ОС, как правило, работают с фиксированным набором программ (функциональных задач). Применение таких систем обусловлено невозможностью использования универсальной ОС по соображениям эффективности, надежности, защищенности и т.п., а также вследствие специфики решаемых задач.   
Универсальные ОС рассчитаны на решение любых задач пользователей, но, как правило, форма эксплуатации вычислительной системы может предъявлять особые требования к ОС, т.е. к элементам ее специализации.

**2.По способу загрузки** можно выделить загружаемые ОС (большинство) и системы, постоянно находящиеся в памяти вычислительной системы. Последние, как правило, специализированные и используются для управления работой специализированных устройств (например, в БЦВМ баллистической ракеты или спутника, научных приборах, автоматических устройствах различного назначения и др.).

**3.По особенностям алгоритмов управления ресурсами**. Главным ресурсом системы является процессор, поэтому дадим классификацию по алгоритмам управления процессором, хотя можно, конечно, классифицировать ОС по алгоритмам управления памятью, устройствами ввода-вывода и.т.д.

**Поддержка многозадачности** (многопрограммности). По числу одновременно выполняемых задач ОС делятся на 2 класса: однопрограммные (однозадачные) – например, MS-DOS, MSX, и многопрограммные (многозадачные) – например, ОС ЕС ЭВМ, OS/360, OS/2, UNIX, Windows разных версий.  
Однопрограммные ОС предоставляют пользователю виртуальную машину, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Они также имеют средства управления файлами, периферийными устройствами и средства общения с пользователем. Многозадачные ОС, кроме того, управляют разделением совместно используемых ресурсов (процессор, память, файлы и т.д.), это позволяет значительно повысить эффективность вычислительной системы.

**Поддержка многопользовательского режима**. По числу одновременно работающих пользователей ОС делятся: на однопользовательские (MS-DOS, Windows 3х, ранние версии OS/2) и многопользовательские (UNIX, Windows NT/2000/2003/XP/Vista).  
Главное отличие многопользовательских систем от однопользовательских – наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что может быть однопользовательская мультипрограммная система.

**Виды многопрограммной работы**. Специфику ОС во многом определяет способ распределения времени между несколькими одновременно существующими в системе процессами (или потоками). По этому признаку можно выделить 2 группы алгоритмов: не вытесняющая многопрограммность (Windows3.x, NetWare) и вытесняющая многопрограммность (Windows 2000/2003/XP, OS/2, Unix).  
В первом случае активный процесс выполняется до тех пор, пока он сам не отдает управление операционной системе. Во втором случае решение о переключении процессов принимает операционная система. Возможен и такой режим многопрограммности, когда ОС разделяет процессорное время между отдельными ветвями (потоками, волокнами) одного процесса.

**Многопроцессорная обработка**. Важное свойство ОС – отсутствие или наличие средств поддержки многопроцессорной обработки. По этому признаку можно выделить ОС без поддержки мультипроцессирования (Windows 3.x, Windows 95) и с поддержкой мультипроцессирования (Solaris, OS/2, UNIX, Windows NT/2000/2003/XP).  
Многопроцессорные ОС классифицируются по способу организации вычислительного процесса на асимметричные ОС (выполняются на одном процессоре, распределяя прикладные задачи по остальным процессорам) и симметричные ОС (децентрализованная система).

**4.По области использования и форме эксплуатации**. Обычно здесь выделяют три типа в соответствии с использованными при их разработке критериями эффективности:

* системы пакетной обработки (OS/360, OC EC);
* системы разделения времени (UNIX, VMS);
* системы реального времени (QNX, RT/11).

Первые предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Критерий создания таких ОС – максимальная пропуская способность при хорошей загрузке всех ресурсов компьютера. В таких системах пользователь отстранен от компьютера.  
Системы разделения времени обеспечивают удобство и эффективность работы пользователя, который имеет терминал и может вести диалог со своей программой.  
Системы реального времени предназначены для управления техническими объектами (станок, спутник, технологический процесс, например доменный и т.п.), где существует предельное время на выполнение программ, управляющих объектом.

**5.По аппаратной платформе** (типу вычислительной техники), для которой они предназначаются, операционные системы делят на следующие группы.

**Операционные системы для смарт-карт**. Некоторые из них могут управлять только одной операцией, например, электронным платежом. Некоторые смарт-карты являются JAVA-ориентированным и содержат интерпретатор виртуальной машины JAVA. Апплеты JAVA загружаются на карту и выполняются JVM-интерпретатором. Некоторые из таких карт могут одновременно управлять несколькими апплетами JAVA, что приводит к многозадачности и необходимости планирования.

**Встроенные операционные системы**. Управляют карманными компьютерами (lialm OS, Windows CE – Consumer Electronics – бытовая техника), мобильными телефонами, телевизорами, микроволновыми печами и т.п.

**Операционные системы для персональных компьютеров**, например, Windows 9.x, Windows ХР, Linux, Mac OSX и др.

**Операционные системы мини-ЭВМ**, например, RT-11 для PDP-11 – OC реального времени, RSX-11 M для PDP-11 – ОС разделения времени, UNIX для PDP-7.

**Операционные системы мэйнфреймов** (больших машин), например, OS/390, происходящая от OS/360 (IBM). Обычно ОС мэйнфреймов предполагает одновременно три вида обслуживания: пакетную обработку, обработку транзакций (например, работа с БД, бронирование авиабилетов, процесс работы в банках) и разделение времени.

**Серверные операционные системы**, например, UNIX, Windows 2000, Linux. Область применения – ЛВС, региональные сети, Intranet, Internet.

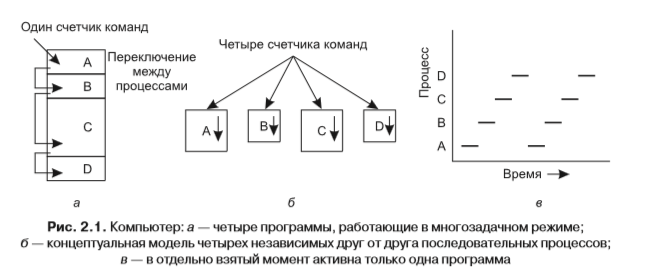
**Кластерные операционные системы**. Кластер – слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений и представляющихся пользователю единой системной, например, Windows 2000 Cluster Server, Windows 2008 Server, Sun Cluster (базовая ОС – Solaris).

1. Процессы (абстракция). Состояния (жизненный цикл) процессов, создание, завершение. Реализация (таблица процессов). Модель многозадачности, оценка утилизации ресурсов процессора, накладные расходы. Процесс как контейнер ресурсов.

Основным понятием в любой операционной системе является процесс: абстракция, описывающая выполняющуюся программу. Процессы — это одна из самых старых и наиболее важных абстракций, присущих операционной системе. Они поддерживают возможность осуществления (псевдо) параллельных операций даже при наличии всего одного центрального процессора. Они превращают один центральный процессор в несколько виртуальных. Без абстракции процессов современные вычисления просто не могут существовать.

Понятие процесса можно трактовать как контейнер ресурсов (выделенная для программ память, файлы, устройства ввода/вывода), а так же как последовательность исполняемых команд.

**Модель процесса**: В этой модели все выполняемое на компьютере программное обеспечение, иногда включая операционную систему, сведено к ряду последовательных процессов, или, для краткости, просто процессов. Процесс — это просто экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных.



Существуют четыре основных события, приводящих к **созданию процессов**.

* Инициализация системы.
* Выполнение работающим процессом системного вызова, предназначенного для создания процесса. (запуск одним процессом другого)
* Запрос пользователя на создание нового процесса.
* Инициация пакетного задания.

С точки зрения техники всегда один процесс запускает другой.

Фоновые процессы, предназначенные для обработки какой-либо активной деятельности, связанной, например, с электронной почтой, веб-страницами, новостями, выводом информации на печать и т. д., называются демонами.  
В UNIX существует только один системный вызов для создания нового процесса — fork. Этот вызов создает точную копию вызывающего процесса. В Windows все происходит иначе: одним вызовом функции Win32 CreateProcess создается процесс, и в него загружается нужная программа.  
В обеих системах, UNIX и Windows, после создания процесса родительский и дочерний процессы обладают своими собственными, отдельными адресными пространствами.

Рано или поздно новые процессы будут завершены, обычно в силу следующих обстоятельств:

* обычного выхода (добровольно);
* выхода при возникновении ошибки (добровольно);
* возникновения фатальной ошибки (принудительно);
* уничтожения другим процессом (принудительно).

(из лекций:

* штатное завершение(программа отработала)
* ошибка программы, после чего происходит штатное завершение
* программа совершила ошибку и была завершена принудительно (не штатно)
* принудительное завершение (не обязательно из-за ошибки) (не штатно) )

Состояния процесса:

* выполняемый (в данный момент использующий центральный процессор);
* готовый (работоспособный, но временно приостановленный, чтобы дать возможность выполнения другому процессу);
* заблокированный (неспособный выполняться, пока не возникнет какое-нибудь внешнее событие).

(из лекций:

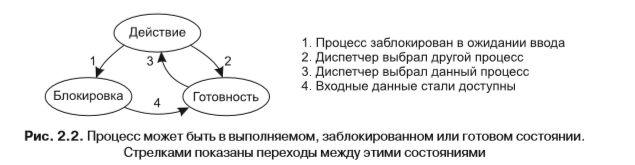
* Исполнения (у процесса есть и данные, и процессор (процессорное время)
* Готовности (данные есть, но процессора не хватает)
* Ожидания (не хватает данных, нет процессора)

Состояние ожидания контролируется ОС (как правило).

При наличии единственного процесса, если тот находится в состоянии ожидания – процессор простаивает. Значит, нужно больше одного процесса.

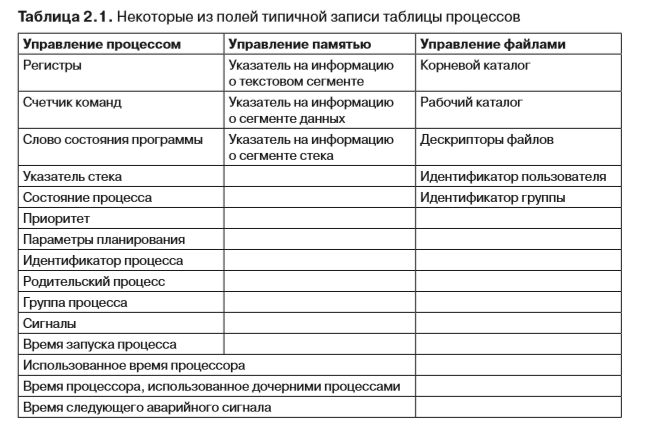
Ожидание->готовность->исполнение

)



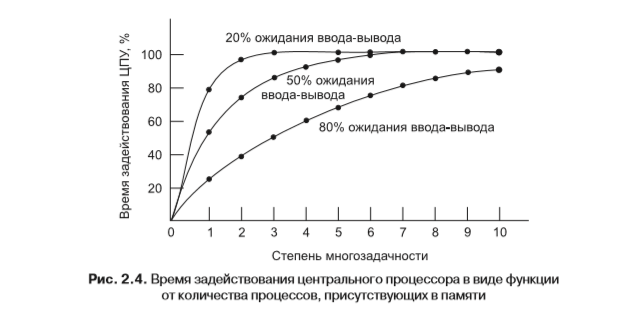
Переход 1 происходит в том случае, если операционная система определит, что процесс в данный момент выполняться не может. В некоторых системах для перехода в заблокированное состояние процесс может осуществить такой системный вызов, как pause.. Переходы 2 и 3 вызываются планировщиком процессов, который является частью операционной системы, без какого-либо оповещения самого процесса. Переход 2 происходит, когда планировщик решит, что выполняемый процесс продвинулся достаточно далеко и настало время позволить другому процессу получить долю рабочего времени центрального процессора. Переход 3 происходит, когда все другие процессы получили причитающуюся им долю времени и настал момент предоставить центральный процессор первому процессу для возобновления его выполнения. Переход 4 осуществляется в том случае, если происходит внешнее событие, ожидавшееся процессом (к примеру, поступление входных данных). Если к этому моменту нет других выполняемых процессов, будет вызван переход 3 и процесс возобновится. В противном случае ему придется немного подождать в состоянии готовности, пока не станет доступен центральный процессор и не придет его очередь.

Для реализации модели процессов операционная система ведет таблицу процессов, в которой каждая запись соответствует какому-нибудь процессу. Эти записи содержат важную информацию о состоянии процесса, включая счетчик команд, указатель стека, распределение памяти, состояние открытых им файлов, его учетную и планировочную информацию и все остальное, касающееся процесса, что должно быть сохранено, когда процесс переключается из состояния выполнения в состояние готовности или блокировки, чтобы позже он мог возобновить выполнение, как будто никогда не останавливался.

Процесс во время своего выполнения может быть прерван тысячи раз, но ключевая идея состоит в том, что после каждого прерывания прерванный процесс возвращается в точности к такому же состоянию, в котором он был до того, как случилось прерывание.

Режим многозадачности позволяет использовать центральный процессор более рационально.

Лучше выстраивать модель на основе вероятностного взгляда на использование центрального процессора. Предположим, что процесс проводит часть своего времени p в ожидании завершения операций ввода-вывода. При одновременном присутствии в памяти n процессов вероятность того, что все n процессов ожидают завершения ввода-вывода (в случае чего процессор простаивает), равна pn. Тогда время задействования процессора вычисляется по формуле Время задействования ценрального процессора = 1 − pn (время задействования центрального процессора в виде функции от аргумента n, который называется степенью многозадачности).



Судя по рисунку, если процесс тратит 80 % своего времени на ожидание завершения ввода-вывода, то для снижения простоя процессора до уровня не более 10 % в памяти могут одновременно находиться по крайней мере 10 процессов (на лекции насчитали 11, и это правильно).

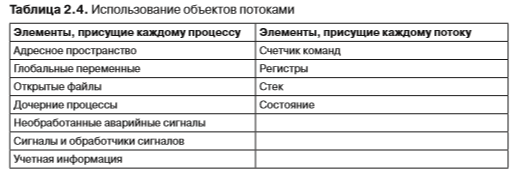
Справедливости ради следует заметить, что рассмотренная нами вероятностная модель носит весьма приблизительный характер. В ней безусловно предполагается, что все n процессов являются независимыми друг от друга, а значит, в системе с пятью процессами в памяти вполне допустимо иметь три выполняемых и два ожидающих процесса. Но имея один центральный процессор, мы не может иметь сразу три выполняемых процесса, поэтому процесс, который становится готовым к работе при занятом центральном процессоре, вынужден ожидать своей очереди. Из-за этого процессы не обладают независимостью.

Несмотря на упрощенность модели, представленной на рис. 2.4, тем не менее она может быть использована для специфических, хотя и весьма приблизительных предсказаний, касающихся производительности центрального процессора. Предположим, к примеру, что память компьютера составляет 8 Гбайт, операционная система и ее таблицы занимают до 2 Гбайт, а каждая пользовательская программа также занимает до 2 Гбайт. Этот объем позволяет одновременно разместить в памяти три пользовательские программы. При среднем ожидании ввода-вывода, составляющем 80 % времени, мы имеем загруженность центрального процессора (если игнорировать издержки на работу операционной системы), равную 1 – 0,83, или около 49 %. Увеличение объема памяти еще на 8 Гбайт позволит системе перейти от трехкратной многозадачности к семикратной, что повысит загруженность центрального процессора до 79 %. Иными словами, дополнительные 8 Гбайт памяти увеличат его производительность на 30 %. Увеличение памяти еще на 8 Гбайт поднимет уровень производительности всего лишь с 79 до 91 %, то есть дополнительный прирост производительности составит только 12 %.

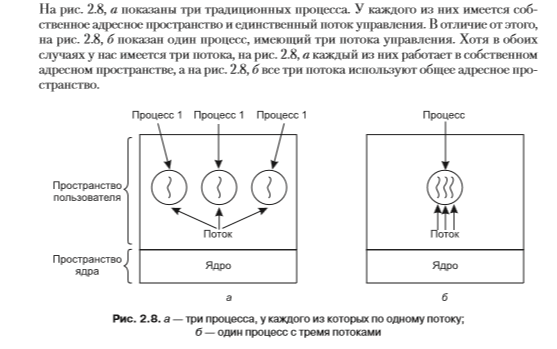
1. Потоки (абстракция). Различные подходы к реализации, сравнение. Возможные проблемы при использовании потоков.

Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме. Рассматривая потоки, мы добавляем новый элемент: возможность использования параллельными процессами единого адресного пространства и всех имеющихся данных. Эта возможность играет весьма важную роль для тех приложений, которым не подходит использование нескольких процессов (с их раздельными адресными пространствами). Вторым аргументом в пользу потоков является легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами( в 10–100 раз быстрее). Это свойство особенно пригодится, когда потребуется быстро и динамично изменять количество потоков. Когда потоки работают в рамках одного центрального процессора, они не приносят никакого прироста производительности, но когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения. И наконец, потоки весьма полезны для систем, имеющих несколько центральных процессоров, где есть реальная возможность параллельных вычислений.

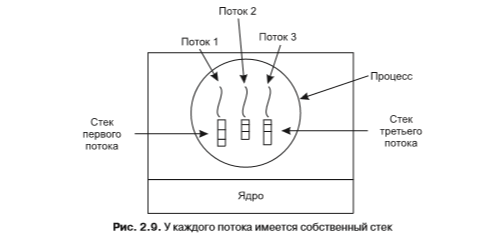
У потока есть счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять. У него есть регистры, в которых содержатся текущие рабочие переменные. У него есть стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры. Хотя поток может быть выполнен в рамках какого-нибудь процесса, сам поток и его процесс являются разными понятиями и должны рассматриваться по отдельности. Процессы используются для группировки ресурсов в единое образование, а потоки являются «сущностью», распределяемой для выполнения на центральном процессоре.



Потоки добавляют к модели процесса возможность реализации нескольких в значительной степени независимых друг от друга выполняемых задач в единой среде процесса. Потоки используют единое адресное пространство и другие ресурсы. Потоки иногда называют облегченными процессами. Термин «многопоточный режим» используется для описания ситуации, при которой допускается работа нескольких потоков в одном и том же процессе.



Когда многопоточный процесс выполняется на однопроцессорной системе, потоки выполняются, сменяя друг друга. Центральный процессор быстро переключается между потоками, создавая иллюзию, что потоки выполняются параллельно, пусть даже на более медленном центральном процессоре, чем реально используемый.



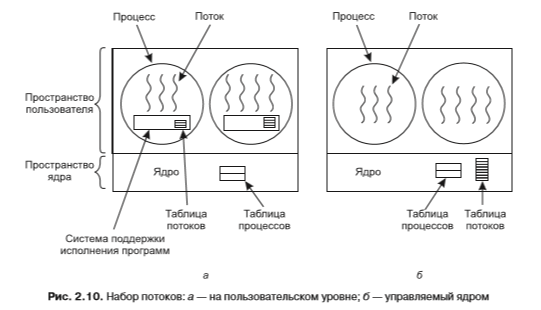
Нет необходимости (или даже возможности) указывать для нового потока какое-нибудь адресное пространство, поскольку он автоматически запускается в адресном пространстве создающего потока. Иногда потоки имеют иерархическую структуру, при которой у них устанавливаются взаимоотношения между родительскими и дочерними потоками, но чаще всего такие взаимоотношения отсутствуют и все потоки считаются равнозначными.

Хотя потоки зачастую приносят пользу, они вносят в модель программирования и ряд сложностей.

Для начала рассмотрим эффект, возникающий при осуществлении системного вызова fork, принадлежащего ОС UNIX. Если у родительского процесса есть несколько потоков, должны ли они быть у дочернего процесса? Если нет, то процесс может неверно функционировать из-за того, что все они составляют его неотъемлемую часть. Но если дочерний процесс получает столько же потоков, сколько их было у родительского процесса, что произойдет, если какой-нибудь из потоков родительского процесса был заблокирован системным вызовом read, используемым, к примеру, для чтения с клавиатуры? Будут ли теперь два потока, в родительском и в дочернем процессах, заблокированы на вводе с клавиатуры? Если будет набрана строка, получат ли оба потока ее копию? Или ее получит только поток родительского процесса? А может быть, она будет получена только потоком дочернего процесса? Сходные проблемы существуют и при открытых сетевых подключениях.

Другая проблема использования потоков: различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У всех потоков абсолютно одно и то же адресное пространство, а значит, они так же совместно используют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому адресу памяти в пределах адресного пространства процесса, один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует, потому что ее невозможно осуществить и в ней нет необходимости. Все потоки, как показано в табл. 2.4, могут совместно использовать одни и те же открытые файлы, дочерние процессы, ожидаемые и обычные сигналы и т. п.   
Что происходит в том случае, если один поток закрывает файл в тот момент, когда другой поток еще не считал с него данные? Предположим, что один поток заметил дефицит свободной памяти и приступил к выделению дополнительного объема. На полпути происходит переключение потоков, и новый поток тоже замечает дефицит свободной памяти и приступает к выделению дополнительного объема. Вполне возможно, что дополнительная память будет выделена дважды. Для решения этих проблем следует приложить ряд усилий, но для корректной работы многопоточных программ требуется все тщательно продумать и спроектировать.

Есть два основных места реализации набора потоков: в пользовательском пространстве и в ядре. Это утверждение носит несколько спорный характер, поскольку возможна еще и гибридная реализация. А теперь мы опишем эти способы со всеми их достоинствами и недостатками.



**Первый способ** — это поместить весь набор потоков в пользовательском пространстве. И об этом наборе ядру ничего не известно. Что касается ядра, оно управляет обычными, однопотоковыми процессами. Первое и самое очевидное преимущество состоит в том, что набор потоков на пользовательском уровне может быть реализован в операционной системе, которая не поддерживает потоки. Когда потоки управляются в пользовательском пространстве, каждому процессу необходимо иметь собственную таблицу потоков, чтобы отслеживать потоки, имеющиеся в этом процессе. Эта таблица является аналогом таблицы процессов, имеющейся в ядре, за исключением того, что в ней содержатся лишь свойства, принадлежащие каждому потоку, такие как счетчик команд потока, указатель стека, регистры, состояние и т. д.  
Потоки имеют те же состояния, что и процессы.  
У потоков, реализованных на пользовательском уровне, есть и другие преимущества. Они позволяют каждому процессу иметь собственные настройки алгоритма планирования.

Но несмотря на лучшую производительность, у потоков, реализованных на пользовательском уровне, есть ряд существенных проблем. Первая из них — как реализовать блокирующие системные вызовы. С проблемой блокирующих системных вызовов несколько перекликается проблема ошибки отсутствия страницы. Использование набора потоков, реализованного на пользовательском уровне, связано еще с одной проблемой: если начинается выполнение одного из потоков, то никакой другой поток, принадлежащий этому процессу, не сможет выполняться до тех пор, пока первый поток добровольно не уступит центральный процессор.  
Т.е., как итог: реализация потоков на пользовательском уровне сложна в реализации и требует большей квалификации разработчика.

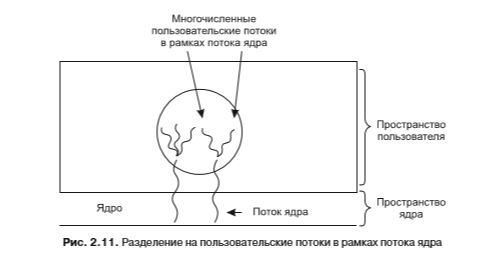
**Второй способ** – если ядро будет знать о потоках и управлять ими.  
В таблице потоков, находящейся в ядре, содержатся регистры каждого потока, состояние и другая информация. Вся информация аналогична той, которая использовалась для потоков, создаваемых на пользовательском уровне, но теперь она содержится в ядре, а не в пространстве пользователя. Эта информация является подмножеством состояния процесса. Вдобавок к этому ядро поддерживает также традиционную таблицу процессов с целью их отслеживания.  
Все вызовы, способные заблокировать поток, реализованы как системные. Когда поток блокируется, ядро по своему выбору может запустить либо другой поток из этого же самого процесса (если имеется готовый к выполнению поток), либо поток из другого процесса. Когда потоки реализуются на пользовательском уровне, система поддержки исполнения программ работает с запущенными потоками собственного процесса до тех пор, пока ядро не заберет у нее центральный процессор (или не останется ни одного готового к выполнению потока).

Для экономии ресурсов, зачастую используются «спящие» потоки: т.е., поток, вместо уничтожения, помечается как неспособный к выполнению, но это не влияет на его структуру данных, имеющуюся в ядре. Чуть позже, когда должен быть создан новый поток, вместо этого повторно активируется старый поток, что приводит к экономии времени.

Для потоков, реализованных на уровне ядра, не требуется никаких новых, неблокирующих системных вызовов. Более того, если один из выполняемых потоков столкнется с ошибкой обращения к отсутствующей странице, ядро может с легкостью проверить наличие у процесса любых других готовых к выполнению потоков и при наличии таковых запустить один из них на выполнение, пока будет длиться ожидание извлечения запрошенной страницы с диска. Главный недостаток этих потоков состоит в весьма существенных затратах времени на системный вызов, поэтому, если операции над потоками (создание, удаление и т. п.) выполняются довольно часто, это влечет за собой более существенные издержки.

Хотя потоки, создаваемые на уровне ядра, и позволяют решить ряд проблем, но справиться со всеми существующими проблемами они не в состоянии. Что будет, к примеру, когда произойдет разветвление многопоточного процесса? Будет ли у нового процесса столько же потоков, сколько у старого, или только один поток? Во многих случаях наилучший выбор зависит от того, выполнение какого процесса запланировано следующим. Если он собирается вызвать команду exec, чтобы запустить новую программу, то, наверное, правильным выбором будет наличие только одного потока. Но если он продолжит выполнение, то лучше всего было бы, наверное, воспроизвести все имеющиеся потоки.

**Третий вариант** - В попытках объединить преимущества создания потоков на уровне пользователя и на уровне ядра была исследована масса различных путей. Один из них (рис. 2.11) заключается в использовании потоков на уровне ядра, а затем нескольких потоков на уровне пользователя в рамках некоторых или всех потоков на уровне ядра. При использовании такого подхода программист может определить, сколько потоков использовать на уровне ядра и на сколько потоков разделить каждый из них на уровне пользователя. Эта модель обладает максимальной гибкостью.



При таком подходе ядру известно только о потоках самого ядра, работу которых оно и планирует. У некоторых из этих потоков могут быть несколько потоков на пользовательском уровне, которые расходятся от их вершины. Создание, удаление и планирование выполнения этих потоков осуществляется точно так же, как и у пользовательских потоков, принадлежащих процессу, запущенному под управлением операционной системы, не способной на многопоточную работу. В этой модели каждый поток на уровне ядра обладает определенным набором потоков на уровне пользователя, которые используют его по очереди.

1. Планирование при многозадачности, задачи. Критерии оценки эффективности, примеры. Вытесняющая и не вытесняющая многозадачности. Приоритеты, оптимальность, справедливость. Планирование потоков и процессов

Основная цель планирования вычислительного процесса заключается в распределении времени процессора (нескольких процессоров) между выполняющимися заданиями пользователей таким образом, чтобы удовлетворять требованиям, предъявляемым пользователями к вычислительной системе.

Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты.  
В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. По разному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ (или же произойдет ошибка, или процесс завершится). В системах с абсолютными приоритетами выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности.

Многозада́чность — свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач. Истинная многозадачность операционной системы возможна только в распределённых вычислительных системах.  
Существует 2 типа многозадачности:

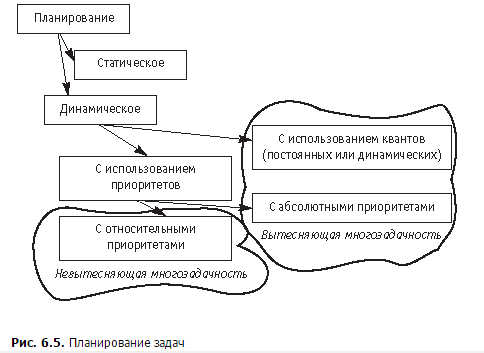
* Процессная многозадачность (основанная на процессах — одновременно выполняющихся программах). Здесь программа — наименьший элемент управляемого кода, которым может управлять планировщик операционной системы.
* Поточная многозадачность (основанная на потоках). Наименьший элемент управляемого кода — поток (одна программа может выполнять 2 и более задачи одновременно).

Многопоточность — специализированная форма многозадачности.

В мультипрограммировании ключевым местом является способ составления расписания, по которому осуществляется переключение между задачами (планирование), а также механизм, осуществляющий эти переключения.

По времени планирования можно выделить статическое и динамическое составление расписания. При статическом планировании расписание составляется заранее, до запуска приложений, и операционная система в дальнейшем просто выполняет составленное расписание. В случае динамического планирования порядок запуска задач и передачи управления задачам определяется непосредственно во время исполнения. Статическое расписание свойственно системам реального времени, когда необходимо гарантировать заданное время и сроки выполнения необходимых операций. В универсальных операционных системах статическое расписание практически не применяется.

(**из лекций** – «Если говорить о приоритетах, то при статической системе приоритет выдается при запуске, и при динамической системе – при долгом ожидании процесса его приоритет повышается». Существует «лотерейное» распределение времени между процессами: каждый процесс имеет «лотерейные билеты». Количество этих «билетов» зависит от приоритета процесса. Чем больше «билетов», тем раньше процессу выделится процессор. В «розыгрыше» участвуют только процессы в состоянии готовности.)



Примитивные многозадачные среды обеспечивают чистое «разделение ресурсов», когда за каждой задачей закрепляется определённый участок памяти, и задача активизируется в строго определённые интервалы времени.

Более развитые многозадачные системы проводят распределение ресурсов динамически, когда задача стартует в памяти или покидает память в зависимости от её приоритета и от стратегии системы. Такая многозадачная среда обладает следующими особенностями:

* Каждая задача имеет свой приоритет, в соответствии с которым получает процессорное время и память
* Система организует очереди задач так, чтобы все задачи получили ресурсы, в зависимости от приоритетов и стратегии системы
* Система организует обработку прерываний, по которым задачи могут активироваться, деактивироваться и удаляться
* По окончании положенного кванта времени ядро временно переводит задачу из состояния выполнения в состояние готовности, отдавая ресурсы другим задачам. При нехватке памяти страницы невыполняющихся задач могут быть вытеснены на диск (своппинг), а потом, через определённое системой время, восстанавливаться в памяти
* Система обеспечивает защиту адресного пространства задачи от несанкционированного вмешательства других задач
* Система обеспечивает защиту адресного пространства своего ядра от несанкционированного вмешательства задач
* Система распознаёт сбои и зависания отдельных задач и прекращает их
* Система решает конфликты доступа к ресурсам и устройствам, не допуская тупиковых ситуаций общего зависания от ожидания заблокированных ресурсов
* Система гарантирует каждой задаче, что рано или поздно она будет активирована
* Система обрабатывает запросы реального времени
* Система обеспечивает коммуникацию между процессами

Основным различием вытесняющего и не вытесняющего вариантов многозадачности является степень централизации механизма планирования процессов. В случае не вытесняющей многозадачности механизм планирования процессов целиком сосредоточен в операционной системе, а в случае вытесняющей многозадачности он распределен между системой и прикладными программами.  
При не вытесняющей многозадачности активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению процесс.  
При вытесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается операционной системой, а не самим активным процессом.

Для невытесняющей многозадачности характерно, что операционная система передает задаче управление и далее ожидает от нее сигнала, информирующего о возможности переключения на другую задачу; сама по себе операционная система выполняемую задачу не прерывает. Именно поэтому невытесняющая многозадачность рассматривается как многозадачность с относительными приоритетами - пока задача сама не сообщит, что настал подходящий для переключения момент, система не сможет передать управление никакой другой, даже высокоприоритетной, задаче.  
Невытесняющая многозадачность проста в реализации, особенно на однопроцессорных машинах, и, кроме того, обеспечивает очень малый уровень накладных расходов на реализацию плана. Недостатками являются повышенная сложность разработки приложений и невысокая защищенность системы от некачественных приложений.  
Характерный **пример невытесняющей многозадачности** - 16-ти разрядные Windows (включая собственно 16-ти разрядные версии Windows, выполнение 16-ти разрядных приложений в Windows-95, 98, ME и выполнение 16-ти разрядных приложений в рамках одной Windows-машины в NT, 2000, XP и 2003). В таких приложениях операционная система не прерывает выполнение текущей задачи до вызова ею функций типа GetMessage или WaitMessage, во время которых Windows осуществляет при необходимости переключение на другую задачу.

Вытесняющая многозадачность предполагает наличие некоторого арбитра, принадлежащего обычно операционной системе, который принимает решение о вытеснении текущей выполняемой задачи какой-либо другой, готовой к выполнению, асинхронно с работой текущей задачи.

В качестве некоторого обобщения можно выделить понятие "момент перепланирования", когда активируется планировщик задач и принимает решение о том, какую именно задачу в следующий момент времени надо начать выполнять. Принципы, по которым назначаются моменты перепланирования, и критерии, по которым осуществляется выбор задачи, определяют способ реализации многозадачности и его сильные и слабые стороны.  
Так, если моменты перепланирования наступают только вследствие явного вызова функций приложением, мы имеем дело с невытесняющей многозадачностью и относительными приоритетами. Если смена приоритета вызывает перепланирование - значит, это система с абсолютными приоритетами и вытесняющей многозадачностью. Если моменты перепланирования наступают по исчерпанию временных квантов (возможно постоянного размера, а возможно и переменного), то система поддерживает вытесняющую многозадачность с квантованием.  
Большинство современных операционных систем используют комбинированные планировщики, одновременно применяющие квантование с переменной продолжительностью кванта и абсолютные или относительные приоритеты.

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести следующие:

* Справедливость – гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.
* Эффективность – постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.
* Сокращение полного времени выполнения– обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
* Сокращение времени ожидания – сократить время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки.
* Сокращение времени отклика – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами.

* Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске.
* Были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.
* Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.
* Обладали масштабируемостью, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, мы ухудшаем ее с точки зрения другого.

1. Оперативная память. Адресация. Адресное пространство (абстракция). Подкачка. Виртуальная память, реализация (таблица). Алгоритмы заполнения и вытеснения страниц.

Та часть операционной системы, которая управляет иерархией памяти (или ее частью), называется менеджером, или диспетчером, памяти. Он предназначен для действенного управления памятью и должен следить за тем, какие части памяти используются, выделять память процессам, которые в ней нуждаются, и освобождать память, когда процессы завершат свою работу

Простейшей абстракцией памяти можно считать полное отсутствие какой-либо абстракции. Ранние универсальные машины (до 1960 года), ранние мини-компьютеры (до 1970 года) и ранние персональные компьютеры (до 1980 года) не использовали абстракции памяти. Каждая программа просто видела физическую память.

При таких условиях содержание в памяти сразу двух работающих программ не представлялось возможным. Если первая программа, к примеру, записывала новое значение в ячейку 2000, то она тем самым стирала то значение, которое сохраняла там вторая программа. Работа становилась невозможной, и обе программы практически сразу же давали сбой

Там ещё написано, что вообще говоря, могли бы работать вместе, если проводился свопинг (как переключение контекстов, тогда программы друг другу не мешают, так как в памяти только исполняемая, остальные выгружаются на диск – это довольно долго) или память помечалась как защищённая и программы не могли друг друга ломать (но так как программа писалась под адресацию с нуля, то вторая программа просто не сможет работать, ибо её адрес первой команды не 0, а какой-нибудь 536, можно бы сделать смещение для всех адресов, но как отличить число от адреса?..). Ещё бы здесь более-менее могли зайти потоки, но абстракция потока есть, а абстракции памяти нет – это слишком странная ситуация.

Вообще говоря, если устройство запускает только конечный и известный набор программ (стиральная машина), то можно обойтись и прямой физической адресацией

**Понятие адресного пространства** Чтобы допустить одновременное размещение в памяти нескольких приложений без создания взаимных помех, нужно решить две проблемы, относящиеся к защите и перемещению. Примитивное решение первой из этих проблем мы уже рассматривали на примере IBM 360: участки памяти помечались защитным ключом, и ключ выполняемого процесса сличался с ключом каждого выбранного слова памяти. Этот подход не решал второй проблемы, хотя она могла быть решена путем перемещения программ в процессе их загрузки, но это было слишком медленным и сложным решением.

Более подходящее решение — придумать для памяти новую абстракцию: адресное пространство. Так же как понятие процесса создает своеобразный абстрактный центральный процессор для запуска программ, понятие адресного пространства создает своеобразную абстрактную память, в которой существуют программы.

Адресное пространство — это набор адресов, который может быть использован процессом для обращения к памяти. У каждого процесса имеется собственное адресное пространство, независимое от того адресного пространства, которое принадлежит другим процессам (за исключением тех особых обстоятельств, при которых процессам требуется совместное использование их адресных пространств).

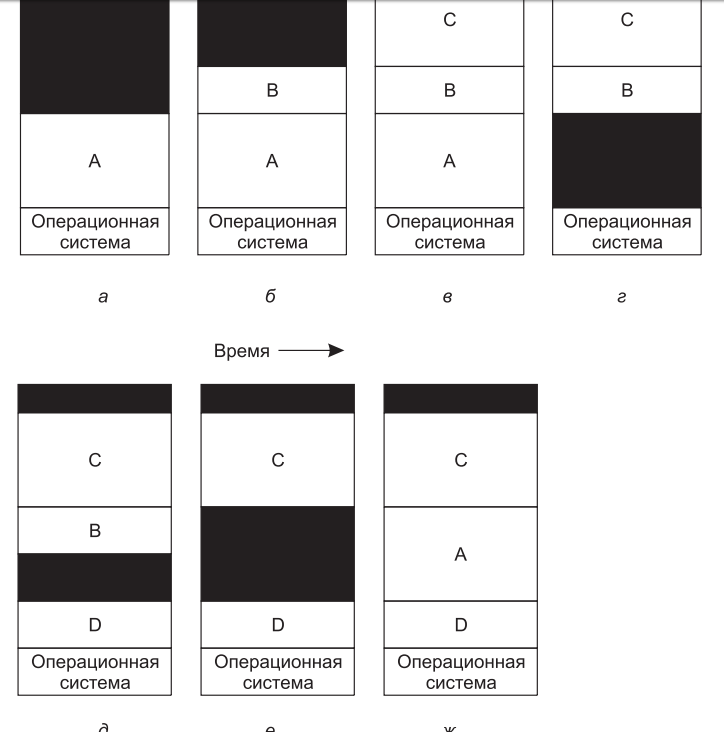
Адресное пространство не обязательно должно быть числовым. Набор интернет-доме- нов .com также является адресным пространством. Это адресное пространство состоит из всех строк длиной от 2 до 63 символов, которые могут быть составлены из букв, цифр и дефисов, за которыми следует название домена — .com. Теперь вам должна стать понятной сама идея, в которой нет ничего сложного.   
Немного сложнее понять, как каждой программе можно выделить собственное адресное пространство, поскольку адрес 28 в одной программе означает иное физическое место, чем адрес 28 в другой программе. Далее мы рассмотрим простой способ, который ранее был распространен, но вышел из употребления с появлением возможностей размещения на современных центральных процессорах более сложных (и более совершенных) схем.

Базовый и ограничительный регистры

В простом решении используется весьма примитивная версия динамического пере- распределения памяти. При этом адресное пространство каждого процесса просто проецируется на различные части физической памяти. Классическое решение, примененное на машинах от CDC 6600 (первого в мире суперкомпьютера) до Intel 8088 (сердца первой модели IBM PC), заключается в оснащении каждого центрального процессора двумя специальными аппаратными регистрами, которые обычно называются базовым и ограничительным регистрами. При использовании этих регистров программы загружаются в последовательно расположенные свободные области памяти без модификации адресов в процессе загрузки. При запуске процесса в базовый регистр загружается физический адрес, с которого начинается, а в ограничительный регистр загружается длина программы. При каждой ссылке процесса на память с целью извлечения команды или записи слова данных аппаратура центрального процессора перед выставлением адреса на шине памяти добавляет к адресу, сгенерированному процессом, значение базового регистра. Одновременно аппаратура проверяет, не равен ли предлагаемый адрес значению ограничительного регистра или не превышает ли он это значение (в этом случае генерируется отказ и доступ прерывается).

Недостатком перемещений с использованием базовых и ограничительных регистров является необходимость применения операций сложения и сравнения к каждой ссылке на ячейку памяти. Сравнение может осуществляться довольно быстро, но сложение является слишком медленной операцией из-за затрат времени на вспомогательный сигнал переноса, если, конечно, не используются специальные сумматоры.

**Своппинг**С годами для преодоления перегрузки памяти были выработаны два основных подхода. Самый простой из них, называемый свопингом, заключается в размещении в памяти всего процесса целиком, его запуске на некоторое время, а затем сбросе на диск. Бездействующие процессы большую часть времени хранятся на диске и в нерабочем состоянии не занимают пространство оперативной памяти (хотя некоторые из них периодически активизируются, чтобы проделать свою работу, после чего опять приостанавливаются). (Второй - виртуальная память – ниже)

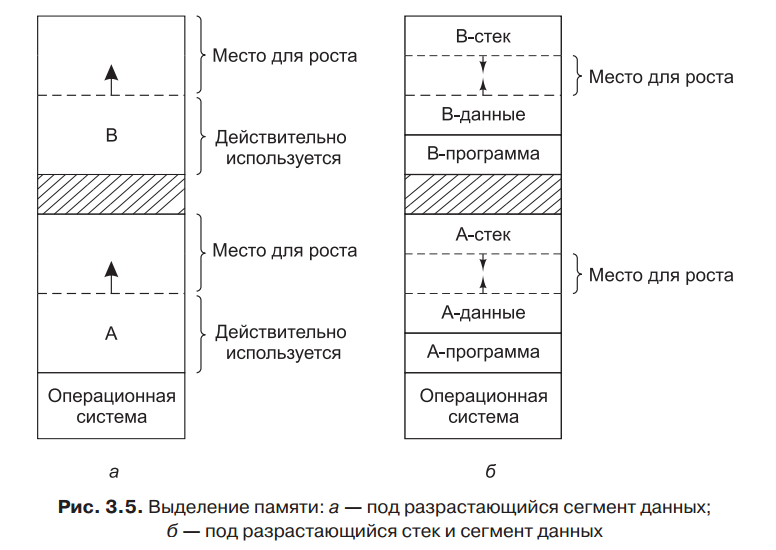


Типа понятная картинка, то что пропало либо завершилось, либо ушло полежать на диск

Когда в результате свопинга в памяти создаются несколько свободных областей, их можно объединить в одну большую за счет перемещения при первой же возможности всех процессов в нижние адреса. Эта технология известна как уплотнение памяти. Но зачастую она не выполняется, поскольку отнимает довольно много процессорного времени.

Но если сегмент данных процесса может разрастаться, к примеру, за счет динамического распределения памяти, как во многих языках программирования, то каждая попытка разрастания процесса вызывает проблему

Поэтому решили выделять процессу побольше, а если ему и этого мало, то после своппинга ему подыщут новое пустое место



**Управление свободной памятью**Если память распределяется в динамическом режиме, то управлять этим должна операционная система. В общих чертах, существуют два способа отслеживания использования памяти: битовые матрицы и списки свободного пространства. (вроде не входит в вопрос – не расписываю)

**Виртуальная память**В то время как для создания абстракции адресного пространства могут быть использованы базовые и ограничительные регистры, нужно решить еще одну проблему: управления ресурсоемким программным обеспечением. Несмотря на быстрый рост объемов памяти, объемы, требующиеся программному обеспечению, растут намного быстрее. В 1980-е годы многие университеты работали на машинах VAX, имеющих память объемом 4 Мбайт, под управлением систем с разделением времени, которые одновременно обслуживали с десяток (более или менее удовлетворенных) пользователей. Теперь корпорация Microsoft рекомендует использовать как минимум 2 Гбайт памяти для 64-разрядной Windows 8. Тенденция к использованию мультимедиа предъявляет к объему памяти еще более весомые требования. Последствия такого развития выразились в необходимости запуска программ, объем которых не позволяет им поместиться в памяти, при этом конечно же возникает потребность в системах, поддерживающих несколько одновременно запущенных программ, каждая из которых помещается в памяти, но все вместе они превышают имеющийся объем памяти. Свопинг — не слишком привлекательный выбор, поскольку обычный диск с интерфейсом SATA обладает пиковой скоростью передачи данных в несколько сотен мегабайт в секунду, а это означает, что свопинг программы объемом 1 Гбайт займет секунды, и еще столько же времени будет потрачено на загрузку другой программы в 1 Гбайт.

В 1960-е годы было принято решение разбивать программы на небольшие части, называемые оверлеями. При запуске программы в память загружался только администратор оверлейной загрузки, который тут же загружал и запускал оверлей с порядковым номером 0. Когда этот оверлей завершал свою работу, он мог сообщить администратору загрузки оверлеев о необходимости загрузки оверлея 1 либо выше оверлея 0, находящегося в памяти (если для него было достаточно пространства), либо поверх оверлея 0 (если памяти не хватало). Некоторые оверлейные системы имели довольно сложное устройство, позволяя множеству оверлеев одновременно находиться в памяти. Оверлеи хранились на диске, и их свопинг с диска в память и обратно осуществлялся администратором загрузки оверлеев. (дело в том, что разбивать на оверлеи должен программист, так и предстваляю, как блин здорово сидеть и думать, в каком месте программу можно разделить на куски)

Изобретенный метод (Fotheringham, 1961) стал известен как виртуальная память. В основе виртуальной памяти лежит идея, что у каждой программы имеется собственное адресное пространство, которое разбивается на участки, называемые страницами. Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы одновременное присутствие в памяти всех страниц необязательно. Когда программа ссылается на часть своего адресного пространства, находящегося в физической памяти, аппаратное обеспечение осуществляет необходимое отображение на лету. Когда программа ссылается на часть своего адресного пространства, которое не находится в физической памяти, операционная система предупреждается о том, что необходимо получить недостающую часть и повторно выполнить потерпевшую неудачу команду

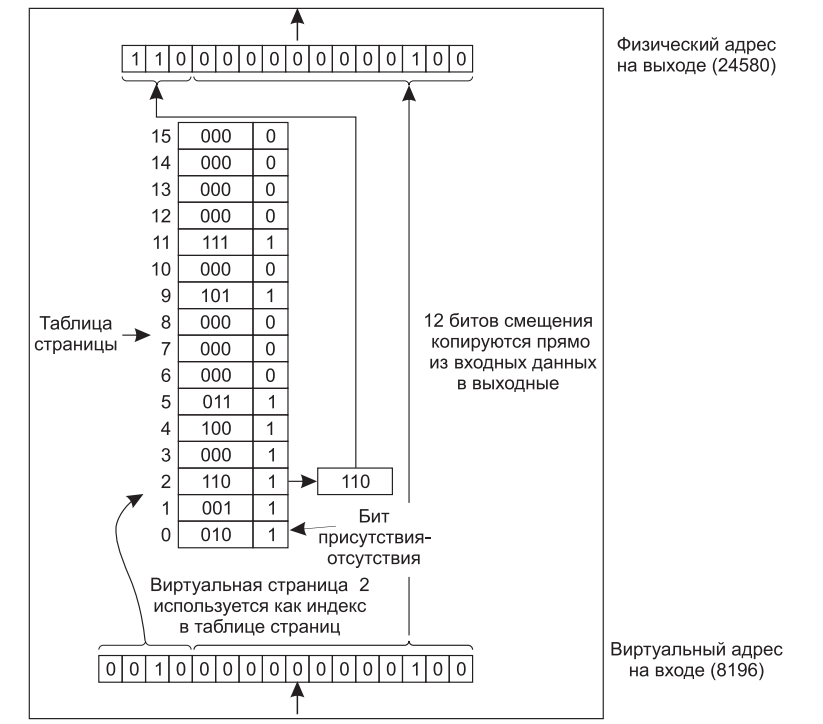
**Страничная организация памяти**В общем, для начала нам надо понять, что оперативной памяти у нас свободно 1Гб, а игре нужно 2, но она реально считает, что у неё их 2, так как создаётся виртуальная область памяти размером в 2Гб, при этом часть её в оперативной памяти, часть на диске (да, игра подвисает – текстурки прогружаются)

Виртуальное адресное пространство состоит из блоков фиксированного размера, называемых страницами. Соответствующие блоки в физической памяти называются страничными блоками. Страницы и страничные блоки имеют, как правило, одинаковые размеры. В реальных системах используются размеры страниц от 512 байт до 1 Гбайт. При наличии 64 Кбайт виртуального адресного пространства и 32 Кбайт физической памяти мы получаем 16 виртуальных страниц и 8 страничных блоков. Перенос информации между оперативной памятью и диском всегда осуществляется целыми страницами. Многие процессоры поддерживают несколько размеров страниц, которые могут быть смешаны и подобраны по усмотрению операционной системы. Например, архитектура x86-64 поддерживает страницы размером 4 Кбайт, 2 Мбайт и 1 Гбайт, поэтому для пользовательских приложений можно использовать страницы размером 4 Кбайт, а для ядра — одну страницу размером 1 Гбайт.  
Далее смотрим на картинку



Сама по себе возможность отображения 16 виртуальных страниц на 8 страничных блоков за счет соответствующей настройки таблиц диспетчера памяти не решает проблемы превышения объема виртуальной памяти над объемом физической памяти. Поскольку в нашем распоряжении только 8 физических страничных блоков, то на физическую память могут отображаться только 8 виртуальных страниц (рис. выше). Остальные, отмеченные на рисунке крестиками, в число отображаемых не попадают. Реальное оборудование отслеживает присутствие конкретных страниц в физической памяти за счет бита присутствия-отсутствия. А что происходит, если, к примеру, программа ссылается на неотображаемые адреса с помощью команды MOV REG,32780 которая обращается к байту 12 внутри виртуальной страницы 8 (которая начинается с адреса 32 768)? Диспетчер памяти замечает, что страница не отображена (поскольку она на рисунке помечена крестиком), и заставляет центральный процессор передать управление операционной системе. Это системное прерывание называется ошибкой отсутствия страницы (page fault). Операционная система выбирает редко используемый страничный блок и сбрасывает его содержимое на диск (если оно еще не там). Затем она извлекает (также с диска) страницу, на которую была ссылка, и помещает ее в только что освободившийся страничный блок, вносит изменения в таблицы и заново запускает прерванную команду.

Ну ещё используются таблицы страниц, примерно так



**Алгоритмы замещения страниц**При возникновении ошибки отсутствия страницы операционная система должна выбрать выселяемую (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы. Если предназначенная для удаления страница за время своего нахождения в памяти претерпела изменения, она должна быть переписана на диске, чтобы привести дисковую копию в актуальное состояние. Но если страница не изменялась (например, она содержала текст программы), дисковая копия не утратила своей актуальности и перезапись не требуется. Тогда считываемая страница просто пишется поверх выселяемой.

Если бы при каждой ошибке отсутствия страницы можно было выбирать для выселе- ния произвольную страницу, то производительность системы была бы намного выше, если бы выбор падал на редко востребуемую страницу. При удалении интенсивно ис- пользуемой страницы высока вероятность того, что она в скором времени будет загру- жена опять, что приведет к лишним издержкам. На выработку алгоритмов замещения страниц было потрачено множество усилий как в теоретической, так и в эксперимен- тальной областях. Далее мы рассмотрим некоторые из наиболее важных алгоритмов.

(кстати проблема очень схожа с проблемой кэша, вроде как тоже полезный буфер)

**Оптимальный алгоритм замещения страниц**

Оптимальный алгоритм замещения страниц гласит, что должна быть удалена страница, имеющая пометку с наибольшим значением. Если какая-то страница не будет использоваться на протяжении 8 млн команд, а другая какая-нибудь страница не будет использоваться на протяжении 6 млн команд, то удаление первой из них приведет к ошибке отсутствия страницы, в результате которой она будет снова выбрана с диска в самом отдаленном будущем. (возникнет позже) Компьютеры, как и люди, пытаются по возможности максимально отсрочить неприятные события. Единственной проблемой такого алгоритма является невозможность его реализации.

**Алгоритм исключения недавно использовавшейся страницы**

Чтобы позволить операционной системе осуществить сбор полезной статистики востребованности страниц, большинство компьютеров, использующих виртуальную память, имеют два бита состояния, R и M, связанных с каждой страницей. Бит R устанавливается при каждом обращении к странице (при чтении или записи). Бит M устанавливается, когда в страницу ведется запись (то есть когда она модифицируется).

При запуске процесса все записи в его таблице страниц помечаются отсутствующими в памяти. Как только произойдет обращение к странице, возникнет ошибка отсутствия страницы. Тогда операционная система устанавливает бит R (в своих внутренних таблицах), изменяет запись в таблице страниц, чтобы она указывала на правильную страницу, с режимом доступа только для чтения (READ ONLY), и перезапускает команду. Если впоследствии страница модифицируется, возникает другая ошибка страницы, позволяющая операционной системе установить бит M и изменить режим доступа к странице на чтение-запись (READ/WRITE). Биты R и M могут использоваться для создания следующего простого алгоритма замещения страниц. При запуске процесса оба страничных бита для всех его страниц устанавливаются операционной системой в 0. Время от времени (например, при каждом прерывании по таймеру) бит R сбрасывается, чтобы отличить те страницы, к которым в последнее время не было обращений, от тех, к которым такие обращения были. При возникновении ошибки отсутствия страницы операционная система просматривает все страницы и на основе текущих значений принадлежащих им битов R и M делит их на четыре категории:

Класс 0: в последнее время не было ни обращений, ни модификаций.

Класс 1: обращений в последнее время не было, но страница модифицирована.

Класс 2: в последнее время были обращения, но модификаций не было.

Класс 3: в последнее время были и обращения, и модификации.

Хотя на первый взгляд страниц класса 1 быть не может, но они появляются в том случае, если у страниц класса 3 бит R сбрасывается по прерыванию от таймера. Эти прерывания не сбрасывают бит M, поскольку содержащаяся в нем информация необходима для того, чтобы узнать, нужно переписывать страницу, хранящуюся на диске, или нет. Сброс бита R без сброса бита M и приводит к возникновению страниц класса 1. Алгоритм исключения недавно использовавшейся страницы (Not Recently Used (NRU)) удаляет произвольную страницу, относящуюся к самому низкому непустому классу. В этот алгоритм заложена идея, суть которой в том, что лучше удалить модифицированную страницу, к которой не было обращений по крайней мере за последний такт системных часов (обычно это время составляет около 20 мс), чем удалить интенсивно используемую страницу. Главная привлекательность алгоритма NRU в том, что его нетрудно понять, сравнительно просто реализовать и добиться от него производительности, которая, конечно, не оптимальна, но может быть вполне приемлема.

**Алгоритм «первой пришла, первой и ушла»**

Очередь страниц, так себе идея, но кто сказал, что нельзя?..

**Алгоритм «второй шанс»**

Простой модификацией алгоритма FIFO, исключающей проблему удаления часто востребуемой страницы, может стать проверка бита R самой старой страницы. Если его значение равно нулю, значит, страница не только старая, но и невостребованная, поэтому она тут же удаляется. Если бит R имеет значение 1, он сбрасывается, а страница помещается в конец списка страниц и время ее загрузки обновляется, как будто она только что поступила в память. Затем поиск продолжается.

Ну да, второй шанс, тут нечего сказать

**Алгоритм «часы»**

При всей своей логичности алгоритм «второй шанс» слишком неэффективен, поскольку он постоянно перемещает страницы в своем списке. Лучше содержать все страничные блоки в циклическом списке в виде часов (рис. 3.15). Стрелка указывает на самую старую страницу.

А смысл тот же, только список циклический

**Алгоритм замещения наименее востребованной страницы**

В основе неплохого приближения к оптимальному алгоритму лежит наблюдение, что страницы, интенсивно используемые несколькими последними командами, будут, скорее всего, снова востребованы следующими несколькими командами. И наоборот, долгое время не востребованные страницы наверняка еще долго так и останутся невостребованными. Эта мысль наталкивает на вполне реализуемый алгоритм: при возникновении ошибки отсутствия страницы нужно избавиться от той страницы, которая длительное время не была востребована. Эта стратегия называется замещением наименее востребованной страницы (Least Recently Used (LRU)). Теоретически реализовать алгоритм LRU вполне возможно, но его практическая реализация дается нелегко. Для его полной реализации необходимо вести связанный список всех страниц, находящихся в памяти. В начале этого списка должна быть только что востребованная страница, а в конце — наименее востребованная. Сложность в том, что этот список должен обновляться при каждом обращении к памяти. Для поиска страницы в списке, ее удаления из него и последующего перемещения этой страницы вперед потребуется довольно много времени, даже если это будет возложено на аппаратное обеспечение (если предположить, что такое оборудование можно создать).

Есть и другие способы – целый параграф Таненбаума

**Алгоритм «рабочий набор»**

Рабочий набор – это такой набор страниц, в котором содержится большинство часто используемых переменных программы (глобальные какие-нибудь переменные и т.п.) и если он не загружен в память, ошибки отсутствия страницы будут возникать очень часто и здорово тормозить процессор

Поэтому многие системы замещения страниц пытаются отслеживать рабочий набор каждого процесса и обеспечивать его присутствие в памяти, перед тем как позволить процессу возобновить работу. Такой подход называется моделью рабочего набора (Denning, 1970). Он был разработан для существенного сокращения количества ошибок отсутствия страниц. Загрузка страниц до того, как процессу будет позволено возобновить работу, называется также опережающей подкачкой страниц (prepaging). Следует заметить, что со временем рабочий набор изменяется.

Ну и алгоритм старается удалить страницу не из рабочего набора (кроме бита R хранится также примерное время последнего обращения).. ладно, подробнее:

Если значение R равно 0, значит, за текущий такт времени обращений к странице не было и она может быть кандидатом на удаление. Чтобы понять, должна ли она быть удалена или нет, вычисляется ее возраст (текущее виртуальное время за вычетом времени последнего использования), который сравнивается со значением t. Если возраст превышает значение t, то страница уже не относится к рабочему набору и заменяется новой страницей. Сканирование продолжается, и происходит обновление всех остальных записей. Но если значение R равно 0, но возраст меньше или равен t, то страница все еще относится к рабочему набору. Страница временно избегает удаления, но страница с наибольшим возрастом (наименьшим значением времени последнего использования) берется на заметку. Если будет просканирована вся таблица страниц и не будет найдена страница — кандидат на удаление, значит, к рабочему набору относятся все страницы. В таком случае, если найдена одна и более страниц с R = 0, удаляется одна из них, имеющая наибольший возраст. В худшем случае в течение текущего такта было обращение ко всем страницам (и поэтому у всех страниц R = 1), поэтому для удаления одна из них выбирается случайным образом, при этом предпочтение отдается неизмененной странице, если таковая имеется.

**Алгоритм WSClock**

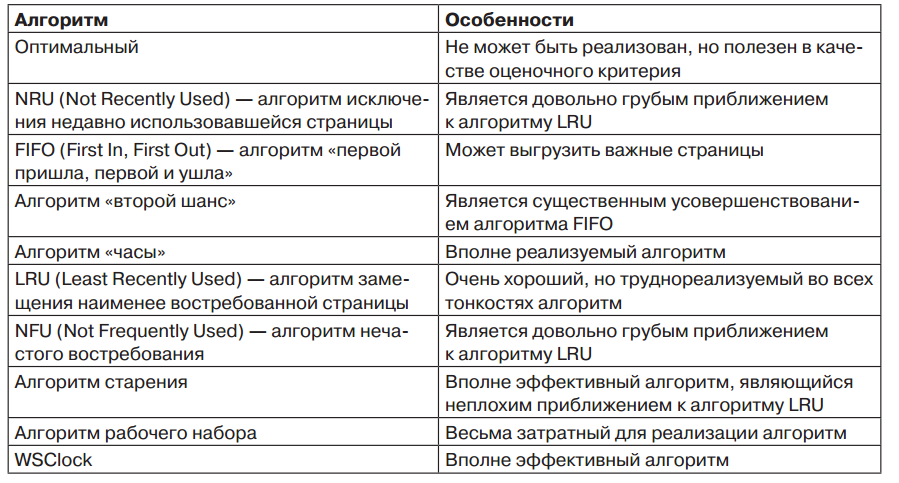
Базовый алгоритм рабочего набора слишком трудоемок, поскольку при возникновении ошибки отсутствия страницы для определения местонахождения подходящего кандидата на удаление необходимо просканировать всю таблицу страниц. Усовершенствованный алгоритм, основанный на алгоритме «часы», но также использующий информацию о рабочем наборе, называется WSClock (Carr and Hennessey, 1981). Благодаря простоте реализации и хорошей производительности он довольно широко используется на практике.

Необходимая структура данных сводится к циклическому списку страничных блоков, как в алгоритме «часы». Изначально этот список пуст. При загрузке первой страницы она добавляется к списку. По мере загрузки следующих страниц они попадают в список, формируя замкнутое кольцо. В каждой записи содержится поле времени последнего использования из базового алгоритма рабочего набора, а также бит R и бит M.

Как и в алгоритме «часы», при каждой ошибке отсутствия страницы сначала проверя- ется страница, на которую указывает стрелка. Если бит R установлен в 1, значит, стра- ница была использована в течение текущего такта, поэтому она не является идеальным кандидатом на удаление. Затем бит R устанавливается в 0, стрелка перемещается на следующую страницу, и алгоритм повторяется уже для нее. Состояние, получившееся после этой последовательности событий, показано на рис. 3.19, б. Теперь посмотрим, что получится, если у страницы, на которую указывает стрелка, бит R = 0 Если ее возраст превышает значение t и страница не изменена, она не относится к рабочему набору и ее точная копия присутствует на диске. Тогда страничный блок просто истребуется и в него помещается новая страница (рис. 3.19, г). Но если страница изменена, ее блок не может быть тотчас же истребован, поскольку на диске нет ее точной копии. Чтобы избежать переключения процесса, запись на диск планируется, а стрелка перемещается дальше и алгоритм продолжает свою работу на следующей странице. В конце концов должна попасться старая, неизмененная страница, которой можно будет тут же и воспользоваться.

А что делать, если стрелка пройдет полный круг и вернется в начальную позицию? Тогда следует рассмотреть два варианта. 1. Была запланирована хотя бы одна запись на диск. 2. Не было запланировано ни одной записи на диск. В первом случае стрелка просто продолжит движение, выискивая неизмененную страницу. Поскольку была запланирована одна или более записей на диск, со вре- менем одна из записей завершится, и задействованная в ней страница будет помечена неизмененной. Первая же неизмененная страница и будет удалена. Эта страница не обязательно должна быть первой запланированной, поскольку драйвер диска может изменить порядок записи, чтобы оптимизировать производительность его работы. Во втором случае все страницы относятся к рабочему набору, иначе должна была быть запланирована хотя бы одна запись. При недостатке дополнительной информации простейшее, что можно сделать, — истребовать любую неизмененную страницу и вос- пользоваться ею. Расположение неизмененной страницы может быть отслежено в про- цессе оборота стрелки. Если неизмененных страниц не имеется, то в качестве жертвы выбирается текущая страница, которая и сбрасывается на диск.

Краткая сравнительная характеристика алгоритмов замещения страниц



1. Носители информации. Файлы и каталоги (абстракции). Структура файловой системы. Развитие атрибутов, прав доступа. Основные операции. Текущий каталог, переменные окружения.

**Носители информации.**К электронным носителям относят носители для однократной или многократной записи:

* оптические (CD-ROM, DVD-ROM, Blu-ray Disc);
* полупроводниковые (флеш-память, дискеты и т. п.).

Жесткий диск состоит из одной или нескольких металлических пластин, вращающихся со скоростью 5400, 7200, 10 800 и более оборотов в минуту. Механический привод поворачивается на определенный угол над пластинами, подобно звукоснимателю старого проигрывателя виниловых пластинок на 33 оборота в минуту. Информация записывается на диск в виде последовательности концентрических окружностей. В каждой заданной позиции привода каждая из головок может считывать кольцеобразный участок, называемый дорожкой. Из совокупности всех дорожек в заданной позиции привода составляется цилиндр.



Все имеющиеся в настоящее время носители информации могут подразделяться по различным признакам. В первую очередь, следует различать энергозависимые и энергонезависимые накопители информации.  
Энергонезависимые накопители, используемые для архивирования и сохранения массивов данных, подразделяют:

1. по виду записи:

* – магнитные накопители (жесткий диск, гибкий диск, сменный диск);
* – магнитно-оптические системы, называемые также МО;
  + оптические, такие, как CD (Compact Disk, Read Only Memory) или DVD (Digital Versatile Disk);

2.по способам построения:

* – вращающаяся пластина или диск (как у жесткого диска, гибкого диска, сменного диска, CD, DVD или MО);
* – ленточные носители различных форматов;
* – накопители без подвижных частей (например, Flash Card, RAM (Random Access Memory), имеющие ограниченную область применения из-за относительно небольших объемов памяти по сравнению с вышеназванными носителями информации).

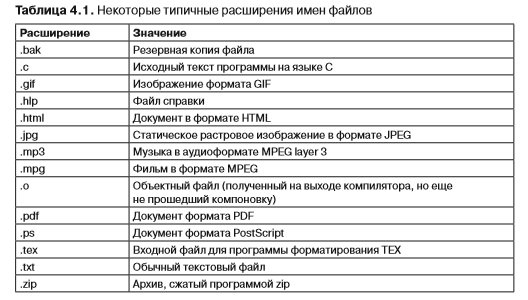
Если требуется быстрый доступ к информации, как, например, при выводе или передаче данных, то используются носители с вращающимся диском. Для архивирования, выполняемого периодически (Backup), наоборот, более предпочтительными являются ленточные носители. Они имеют большие объемы памяти в сочетании с невысокой ценой, правда, при относительно невысоком быстродействии.

3.По назначению носители информации различаются на три группы:

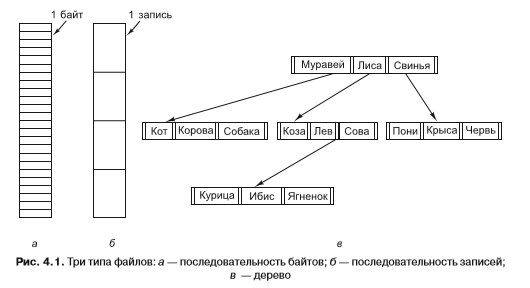
* распространение информации: носители с предварительно записанной информацией, такие как CD ROM или DVD-ROM;
* архивирование: носители для одноразовой записи информации, такие как CD-R или DVD-R (R (record able) – для записи);
* резервирование (Backup) или передача данных: носители с возможностью многоразовой записи информации, такие как дискеты, жесткий диск, MO, CD-RW (RW (rewritable) – перезаписываемые и ленты.

**Общая информация по файлам.**Ключевым понятием операционной системы является файловая система. Файл является механизмом абстрагирования. Он предоставляет способ сохранения информации на диске и последующего ее считывания, который должен оградить пользователя от подробностей о способе и месте хранения информации и деталей фактической работы дисковых устройств.

Когда процесс создает файл, он присваивает ему имя. Когда процесс завершается, файл продолжает существовать, и к нему по этому имени могут обращаться другие процессы. Конкретные правила составления имен файлов варьируются от системы к системе (Windows 95 и Windows 98, использовали файловую систему MS-DOS под названием FAT-16). Та часть имени, которая следует за точкой, называется расширением имени файла и, как правило, несет в себе некоторую информацию о файле.



Файлы могут быть структурированы несколькими различными способами

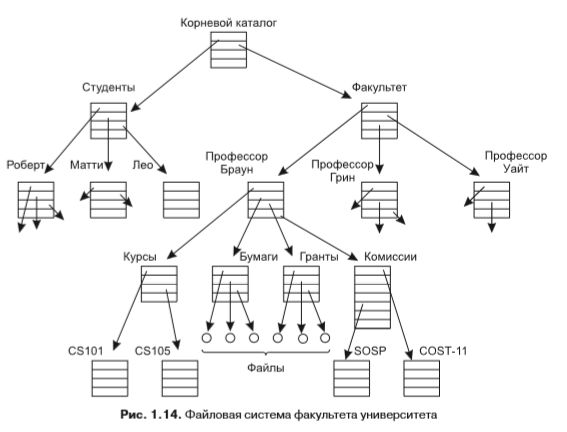


Многие операционные системы поддерживают несколько типов файлов. К примеру, в системах UNIX (опять же включая OS X) и Windows имеются обычные **файлы** **и каталоги**. В системе UNIX имеются также **символьные и блочные специальные файлы**. Обычными считаются файлы, содержащие информацию пользователя. Все файлы на рис. 4.1 являются обычными.

Файл состоит из пяти разделов: заголовка, текста, данных, битов перемещения и таблицы символов. Заголовок начинается с так называемого магического числа, идентифицирующего файл в качестве исполняемого (чтобы предотвратить случайное исполнение файла, не соответствующего данному формату). Затем следуют размеры различных частей файла, адрес, с которого начинается его выполнение, и ряд битов-флагов. За заголовком следуют текст программы и данные. Они загружаются в оперативную память и перемещаются с использованием битов перемещения. Таблица символов используется для отладки.

Обычно в файловой системе для упорядочения файлов имеются каталоги или папки, которые сами по себе являются файлами. Каталоги — это системные файлы, предназначенные для поддержки структуры файловой системы. Мы рассмотрим их чуть позже. Символьные специальные файлы имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, к которым относятся терминалы, принтеры и сети. Блочные специальные файлы используются для моделирования дисков. В данной главе нас в первую очередь будут интересовать обычные файлы.

Эта модель стала прообразом иерархической структуры файловой системы, один из вариантов которой показан на рис. 1.14.

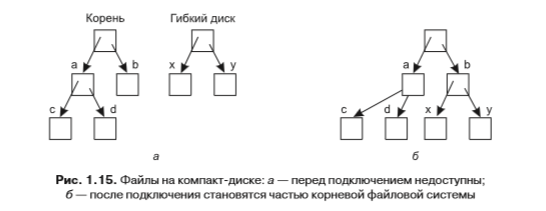


Иерархии файлов организованы в виде деревьев. Иерархии файлов обычно имеют глубину в четыре, пять и более уровней. Иерархия каталогов может существовать годами. Практически всегда существуют механизмы, позволяющие читать файлы и каталоги не только их владельцу, но и более широкой группе пользователей.

Каждый файл, принадлежащий иерархии каталогов, может быть обозначен своим полным именем с указанием пути к файлу, начиная с вершины иерархии — корневого каталога. Этот абсолютный путь состоит из списка каталогов, которые нужно пройти от корневого каталога, чтобы добраться до файла.

Другой разновидностью имени является относительное имя. Оно используется совместно с понятием рабочего каталога (называемого также текущим каталогом). В любой момент времени у каждого процесса есть текущий рабочий каталог, относительно которого рассматриваются пути файлов, не начинающиеся с косой черты. Пользователь может определить один каталог в качестве текущего, и тогда все имена файлов станут рассматриваться относительно рабочего каталога и не будут начинаться с корневого каталога.

Важное понятие в UNIX — смонтированная файловая система. Чтобы предоставить удобный способ работы с съемными носителями информации, UNIX позволяет файловой системе на оптическом диске подключаться к основному дереву.   
Перед вызовом команды mount корневая файловая система на жестком диске и вторая файловая система на компакт-диске существуют отдельно и не связаны друг с другом. Однако файлы на компакт-диске нельзя использовать, поскольку отсутствует способ определения для них полных имен. UNIX не позволяет указывать в начале полного имени номер или имя устройства. Вместо этого системный вызов mount позволяет подключить файловую систему на компакт-диске к корневой файловой системе в том месте, где этого потребует программа. На рис. 1.15, б файловая система на компакт-диске была подключена к каталогу b, открыв доступ к файлам /b/x и /b/y. Если в каталоге b содержались какие-нибудь файлы, то пока к нему подключена файловая система компакт-диска, эти файлы будут недоступны, поскольку путь /b стал ссылкой на корневой каталог компакт-диска. Если система оснащена несколькими жесткими дисками, то все они могут быть подключены к единому дереву аналогичным образом.



Канал — это разновидность псевдофайла, которым можно воспользоваться для соединения двух процессов. Когда процессу A нужно отправить данные процессу B, он осуществляет запись в канал, как будто имеет дело с выходным файлом. Фактически реализация канала очень похожа на реализацию файла. Процесс B может прочитать данные, осуществляя операцию чтения из канала, как будто он имеет дело с входным файлом.

**Атрибуты файла.**У каждого файла есть свои имя и данные. Вдобавок к этому все операционные системы связывают с каждым файлом и другую информацию, к примеру дату и время последней модификации файла и его размер. Мы будем называть эти дополнительные сведения атрибутами файла. Также их называют метаданными. Список атрибутов существенно варьируется от системы к системе.  
В табл. 4.2 показаны некоторые из возможных атрибутов, но кроме них существуют и другие атрибуты. Ни одна из существующих систем не имеет всех этих атрибутов, но каждый из них присутствует в какой-либо системе. Первые четыре атрибута относятся к защите файла и сообщают о том, кто может иметь к нему доступ, а кто нет. Флаги представляют собой биты или небольшие поля, с помощью которых происходит управление некоторыми конкретными свойствами или разрешение их применения. Поля длины записи, позиции ключа и длины ключа имеются только у тех файлов, записи которых можно искать по ключу. Они предоставляют информацию, необходимую для поиска ключей. Различные показатели времени позволяют отслеживать время создания файла, последнего доступа к этому файлу, его последнего изменения. Текущий размер показывает, насколько большим является файл в настоящее время. Некоторые старые операционные системы универсальных машин требуют при создании файла указывать его максимальный размер, чтобы позволить операционной системе заранее выделить максимальное место для его хранения. Операционные системы рабочих станций и персональных компьютеров достаточно разумны, чтобы обойтись без этой особенности.



**Операции с файлами.**Вполне очевидно, что для создания, удаления, чтения и записи файлов понадобятся системные вызовы. Перед тем как файл будет готов к чтению, он должен быть найден на диске и открыт, а после считывания — закрыт. Рассмотрим наиболее распространенные системные вызовы, относящиеся к работе с файлами.

* Create (Создать). Создает файл без данных. Цель вызова состоит в объявлении о появлении нового файла и установке ряда атрибутов.
* Delete (Удалить). (чтобы освободить дисковое пространств)
* Open (Открыть). Перед использованием файла процесс должен его открыть. Цель системного вызова open — дать возможность системе извлечь и поместить в оперативную память атрибуты и перечень адресов на диске, чтобы ускорить доступ к ним при последующих вызовах.
* Close (Закрыть). ( чтобы освободить место во внутренней таблице
* Read (Произвести чтение). Считывание данных из файла
* Write (Произвести запись). Запись данных в файл, как правило, с текущей позиции. Если эта позиция находится в конце файла, то его размер увеличивается. Если текущая позиция находится где-то в середине файла, то новые данные пишутся поверх существующих, которые утрачиваются навсегда.
* Append (Добавить данные в конец файла)
* Seek (Найти). ( перемещает указатель файла к определенной позиции в файле
* Get attributes (Получить атрибуты
* Set attributes (Установить атрибуты
* Rename (Переименовать).

**Основные операции с каталогами.**

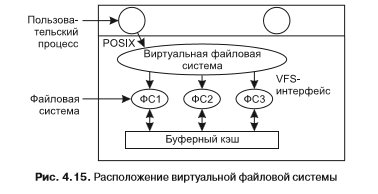
* Create (Создать каталог)
* Delete (Удалить каталог)
* Opendir (Открыть каталог)
* Closedir (Закрыть каталог)
* Readdir (Прочитать каталог). Этот вызов возвращает следующую запись из открытого каталога.
* Rename (Переименовать каталог).
* Link (Привязать). Привязка представляет собой технологию, позволяющую файлу появляться более чем в одном каталоге. В этом системном вызове указываются существующий файл и новое имя файла в некотором существующем каталоге и создается привязка существующего файла к указанному каталогу с указанным новым именем. Таким образом, один и тот же файл может появиться в нескольких каталогах, возможно, под разными именами. Подобная привязка, увеличивающая показания файлового счетчика i-узла (предназначенного для отслеживания количества записей каталогов, в которых фигурирует файл), иногда называется жесткой связью, или жесткой ссылкой (hard link).
* Unlink (Отвязать). Удалить запись каталога. Если отвязываемый файл присутствует только в одном каталоге (что чаще всего и бывает), то этот вызов удалит его из файловой системы

**Структура файловой системы.**Файловые системы хранятся на дисках. Большинство дисков может быть разбито на один или несколько разделов, на каждом из которых будет независимая файловая система.  
Сектор 0 на диске называется главной загрузочной записью (Master Boot Record (MBR)) и используется для загрузки компьютера. В конце MBR содержится таблица разделов. Из этой таблицы берутся начальные и конечные адреса каждого раздела. Один из разделов в этой таблице помечается как активный. При загрузке компьютера BIOS (базовая система ввода-вывода) считывает и выполняет MBR. Первое, что делает программа MBR, — находит расположение активного раздела, считывает его первый блок, который называется загрузочным, и выполняет его. Программа в загрузочном блоке загружает операционную систему, содержащуюся в этом разделе. Зачастую файловая система будет содержать некоторые элементы, показанные на рис. 4.6. Первым элементом является суперблок. В нем содержатся все ключевые параметры файловой системы, которые считываются в память при загрузке компьютера или при первом обращении к файловой системе. Обычно в информацию суперблока включаются «магическое» число, позволяющее идентифицировать тип файловой системы, количество блоков в файловой системе, а также другая важная административная информация. Далее может находиться информация о свободных блоках файловой системы, к примеру, в виде битового массива или списка указателей. За ней могут следовать i-узлы, массив структур данных — на каждый файл по одной структуре, в которой содержится вся информация о файле. Затем может размещаться корневой каталог, содержащий вершину дерева файловой системы. И наконец, оставшаяся часть диска содержит все остальные каталоги и файлы.

**Переменные окружения.**Переменные среды́(окружения) — текстовые переменные операционной системы, хранящие данные о ряде настроек операционной системы.

1. Промежуточные уровни абстракции: менеджер логических томов и виртуальная файловая система. Специализация файловых систем. Отказоустойчивость файловых систем (дублирование метаинформации, журналирование, распределение копий данных). Направления развития.

**Виртуальные файловые системы.**Существование нескольких файловых систем становится необходимостью, и начиная с передовой разработки Sun Microsystems (Kleiman, 1986) большинство UNIX-систем, пытаясь интегрировать несколько файловых систем в упорядоченную структуру, использовали концепцию виртуальной файловой системы (virtual file system (VFS)). Ключевая идея состоит в том, чтобы выделить какую-то часть файловой системы, являющуюся общей для всех файловых систем, и поместить ее код на отдельный уровень, из которого вызываются расположенные ниже конкретные файловые системы с целью фактического управления данными. Вся структура показана на рис. 4.15.

  
Все относящиеся к файлам системные вызовы направляются для первичной обработки в адрес виртуальной файловой системы. Эти вызовы, поступающие от пользовательских процессов, являются стандартными POSIX-вызовами, такими как open, read, write, lseek и т. д. Таким образом, VFS обладает «верхним» интерфейсом к пользовательским процессам, и это хорошо известный интерфейс POSIX. У VFS есть также «нижний» интерфейс к конкретной файловой системе, который на рис. 4.15 обозначен как VFS-интерфейс. Этот интерфейс состоит из нескольких десятков вызовов функций, которые VFS способна направлять к каждой файловой системе для достижения конечного результата.  
Таким образом, у VFS имеются два интерфейса: «верхний» — к пользовательским процессам и «нижний» — к конкретным файловым системам.

По внутреннему устройству большинство реализаций VFS являются объектно-ориентированными. Как правило, в них поддерживается ряд ключевых типов объектов. Среди них суперблок (superblock), описывающий файловую систему, v-узел (v-node), описывающий файл, и каталог (directory), описывающий каталог файловой системы. Каждый из них имеет связанные операции (методы), которые должны поддерживаться конкретной файловой системой. Вдобавок к этому в VFS имеется ряд внутренних структур данных для собственного использования, включая таблицу монтирования и массив описателей файлов, позволяющий отслеживать все файлы, открытые в пользовательских процессах.

Таким образом, виртуальная файловая система — уровень абстракции поверх конкретной реализации файловой системы. Целью VFS является обеспечение единообразного доступа клиентских приложений к различным типам файловых систем. VFS может быть использована, например, для прозрачного доступа к локальным и сетевым устройствам хранения данных без использования специального клиентского приложения (независимо от типа файловой системы). VFS определяет интерфейс между ядром и конкретной файловой системой, таким образом, можно легко добавлять поддержку новых типов файловых систем, внося изменения только в ядро операционной системы.

**Менеджер логических томов.**Менеджер логических томов (Logical-Volume Manager - LVM)- это утилита, которая позволяет управлять дисковым пространством путем использования заданных пользователем логических томов.  
LVM добавляет уровень абстракции между физическими/логическими дисками (привычными разделами, с которыми работает fdisk и аналогичные программы) и файловой системой. Это достигается путём разбивки изначальных разделов на блоки, либо использования отдельных разделов или блочных устройств (physical volume (pv)) и объединения их в единый виртуальный том, точнее группу томов (volume group (vg)), которая далее разбивается на логические тома (logical volume (lv)). Для файловой системы логический том представлен как обычное блочное устройство, хотя отдельные pv тома могут находиться на разных физических устройствах (и даже сам pv может быть распределён подобно RAID).  
LVM1 не поддерживает расширение существующего логического тома на новые физические тома. LVM2 позволяет это, но не отменяет ограничений, связанных с реализацией файловых систем (ФС).

**Специализация файловых систем**По предназначению файловые системы можно классифицировать на нижеследующие категории.

* Для носителей с произвольным доступом (например, жёсткий диск): FAT32, HPFS, ext2 и др. Поскольку доступ к дискам в несколько раз медленнее, чем доступ к оперативной памяти, для прироста производительности во многих файловых системах применяется асинхронная запись изменений на диск. Для этого применяется либо журналирование, например в ext3, ReiserFS, JFS, NTFS, XFS, либо механизм soft updates и др. Журналирование широко распространено в Linux, применяется в NTFS. Soft updates — в BSD системах.
* Для носителей с последовательным доступом (например, магнитные ленты): QIC и др.
* Для оптических носителей — CD и DVD: ISO9660, HFS, UDF и др.
* Виртуальные файловые системы: AEFS и др.
* Сетевые файловые системы: NFS, CIFS, SSHFS, GmailFS и др.
* Для флэш-памяти: YAFFS, ExtremeFFS, exFAT.
* Немного выпадают из общей классификации специализированные файловые системы: ZFS (собственно файловой системой является только часть ZFS), VMFS (т. н. кластерная файловая система, которая предназначена для хранения других файловых систем) и др.

(ZFS (Zettabyte File System) – это 128-битная файловая система, которая поддерживает файлы до смешного огромных размеров (16 эксабайт) и способна работать с дисковыми объемами до 256 зеттабайт.. Файловая система Btrfs является прямым конкурентом ZFS и обладает практически теми же функциями.)

**Отказоустойчивость файловых систем**.  
Жизнь полна неприятных неожиданностей, а разрушение файловой системы зачастую более опасно, чем разрушение компьютера. Поэтому файловые системы должны разрабатываться с учетом подобной возможности. Помимо очевидных решений, например своевременное дублирование информации (backup), файловые системы современных ОС содержат специальные средства для поддержки собственной совместимости.

**Журналирование**  
Другим средством поддержки целостности является заимствованный из систем управления базами данных прием, называемый журнализация (иногда употребляется термин "журналирование" ). Последовательность действий с объектами во время файловой операции протоколируется, и если произошел останов системы, то, имея в наличии протокол, можно осуществить откат системы назад в исходное целостное состояние, в котором она пребывала до начала операции. Подобная избыточность может стоить дорого, но она оправдана, так как в случае отказа позволяет реконструировать потерянные данные.  
Для отката необходимо, чтобы для каждой протоколируемой в журнале операции существовала обратная. Например, для каталогов и реляционных СУБД это именно так. По этой причине, в отличие от СУБД, в файловых системах протоколируются не все изменения, а лишь изменения метаданных ( индексных узлов, записей в каталогах и др.). Изменения в данных пользователя в протокол не заносятся. Кроме того, если протоколировать изменения пользовательских данных, то этим будет нанесен серьезный ущерб производительности системы, поскольку кэширование потеряет смысл.  
Журнализация реализована в NTFS, Ext3FS, ReiserFS и других системах. Чтобы подчеркнуть сложность задачи, нужно отметить, что существуют не вполне очевидные проблемы, связанные с процедурой отката. Например, отмена одних изменений может затрагивать данные, уже использованные другими файловыми операциями. Это означает, что такие операции также должны быть отменены. Данная проблема получила название каскадного отката транзакций

**Дублирование метаинформации**Метаданные, метаинформация— это данные о данных, которые описывают характеристики объектов-носителей данных, способствуют обнаружению, идентификации, оценке и управлению этими данными, включают определения объектов, относящихся к данным, разработчикам, пользователям и средствам взаимодействия.

Файловые системы с журнализацией разделяются на две категории: системы, производящие журнализацию всех изменений, и системы, журнализирующие только изменения метаданных – служебной информации о жестком диске и информация о структуре разделов файловой системы. Это информация о данных на диске. Таким образом, при возникновении ошибок в работе системы, возможен её откат и полное восстановление данных с помощью имеющихся в распоряжении метаданных дисков. Поэтому дублирование такого рода информации позволяет значительно повысить надежность работы.

**Перспективы развития**

На данный момент специалисты выделяют 2 технологии, способные привести к развитию новых видов файловых систем:

* Технология увеличения дискового пространства (Черепичная магнитная запись, Shindled magnetic recording, SMR)
* Создание персистентных систем (persistent memory)

Указанные технологии радикально изменят системы хранения данных. Определяющим моментом для разработчиков, будет необходимость найти более-менее приличный вариант работы существующих файловых систем поверх SMR и/или устройств "persistent memory" или создать новый. Однажды кто-то предложит ФС, которая будет предназначена для этих систем, и она сметет все другие с рынка.

1. Виртуализация. Типы виртуализации, применение. Облако как продукт виртуализации

(то, что выделено таким цветом – больше вода. Совсем сжатая суть написана нормально, чёрным)

Виртуализация позволяет одному компьютеру стать базой для нескольких виртуальных машин, на каждой из которых потенциально может быть запущена совершенно другая операционная система. Преимуществом такого подхода является то, что авария одной виртуальной машины не приводит к аварии любой другой машины. Более того, теперь на одном и том же оборудовании можно запускать несколько разных операционных систем, получая преимущества изолированности виртуальных машин при угрозе хакерских атак и другие преимущества.

Однако, при падении сервера, на котором запущены все эти виртуальные машины, результат будет еще катастрофичнее падения отдельного выделенного сервера. Но смысл связываться с виртуализацией заключается в том, что подавляющая часть выходов служб из строя происходит не по вине оборудования, а из-за недочетов в разработке, ненадежности, наличия ошибок и плохой настройки программного обеспечения, включая, и это следует особо подчеркнуть, операционные системы. При использовании технологии виртуальных машин единственным программным обеспечением, запущенным в режиме наивысших привилегий, является гипервизор, у которого имеется на два порядка меньше строк кода, чем у всей операционной системы, а, следовательно, и на два порядка меньше потенциальных ошибок. Гипервизор проще операционной системы, поскольку он занимается только одним — эмулированием нескольких копий оборудования.

Виртуализация также содействует проверке жизнеспособности новых идей. Обычно в крупных компаниях отдельные подразделения или группы занимаются проработкой интересных идей, а затем идут на затраты, приобретая сервер для их реализации. Если идея получает популярность и ей необходимы сотни или тысячи серверов, дата-центр корпорации расширяется. Зачастую перемещение программного обеспечения на уже существующие машины дается нелегко, поскольку каждому приложению часто требуется другая версия операционной системы, его собственные библиотеки, конфигурационный файлы и многое другое. При использовании виртуальных машин каждое приложение может взять с собой все свое окружение.

Еще одним преимуществом виртуальных машин является то, что установка контрольных точек и миграция этих виртуальных машин (например, для выравнивания баланса загруженности нескольких серверов) даются намного легче, чем миграция процессов, запущенных на обычной операционной системе.

Еще одним важным аспектом использования виртуальных машин является разработка программного обеспечения. Программист, желающий убедиться в работоспособности своей программы под Windows 7, Windows 8, несколькими версиями Linux, FreeBSD, OpenBSD, NetBSD и OS X, а также под управлением других систем, теперь не нуждается в десятке компьютеров и в установке операционных систем на все эти компьютеры. Вместо этого он просто создает десять виртуальных машин на одном компьютере и устанавливает на каждую из них разные операционные системы.

Возможно, наиболее важным и соответствующим времени случаем использования виртуализации является **облако** (cloud). Ключевая идея облака довольно проста: передать ваши потребности в вычислениях или хранении данных в высокоорганизованный дата-центр, запущенный компанией, специализирующейся на подобных услугах и укомплектованной специалистами в данной области. Поскольку изолированность обеспечивается виртуализацией, поставщики облачных услуг могут позволить нескольким клиентам, даже конкурирующим друг с другом, пользоваться общей физической машиной.

**История**По меркам Интернета виртуальные машины являются весьма древними устройствами. Их истоки теряются в 60-х годах прошлого столетия. В IBM проводились эксперименты даже не с одним, а с двумя разработанными независимо друг от друга гипервизорами: SIMMON и CP-40. В 1974 году двое ученых из Калифорнийского университета, Геральд Попек и Роберт Голдберг, опубликовали основополагающую статью, в которой дали точный перечень тех условий, которым должна отвечать компьютерная архитектура, чтобы иметь возможность эффективно поддерживать виртуализацию.

В 1990-е годы исследователи из Стэндфордского университета разработали новый гипервизор с именем Disco и стали основателями VMware, гиганта виртуализации, предлагающего гипервизоры типа 1 и типа. Кстати, разница между гипервизорами типа 1 и типа 2 также пришла из 1970-х. VMware представила свое первое решение по виртуализации для x86 в 1999 году. Затем последовали и другие продукты: Xen, KVM, VirtualBox, Hyper-V, Parallels и многие другие. Похоже, время для виртуализации как раз подоспело, даже при том, что теорию застолбили еще в 1974 году, а IBM десятилетиями продавала компьютеры, поддерживающие и активно использующие виртуализацию. В 1999 году виртуализация приобрела широкую популярность, но несмотря на внезапно возросший к ней массовый интерес, новинкой она не была.

Важно понимать, что виртуальные машины работают так же, как и реальные. В частности, у них должна быть возможность начальной загрузки, как на реальных машинах, и установки на них произвольных операционных систем, точно так же, как это может быть сделано на реальном оборудовании. Предоставление этой иллюзии с обеспечением достаточной эффективности является задачей гипервизора. Несомненно, гипервизоры должны хорошо проявлять себя по трем направлениям:

* Безопасность — у гипервизора должно быть полное управление виртуализированными ресурсами.
* Эквивалентность — поведение программы на виртуальной машине должно быть идентичным поведению этой же программы, запущенной на реальном оборудовании.
* Эффективность — основная часть кода в виртуальной машине должна выполняться без вмешательства гипервизора.

Несомненно, безопасный способ выполнения инструкций заключается в поочередном рассмотрении каждой инструкции в интерпретаторе (например, в Bochs) и в выполнении именно того, что нужно для данной инструкции. Например, нельзя разрешать гостевой операционной системе блокировать прерывания для всей машины или модифицировать отображения страниц в таблицах. Нужно применить прием, заставляющий операционную систему, посаженную поверх гипервизора, полагать, что она заблокировала прерывания или изменила отображение страниц на машине.

Виртуализация долгое время была проблемой для архитектуры x86 из-за дефектов в архитектуре Intel 386. В двух словах, каждый центральный процессор с режимом ядра и пользовательским режимом имеет набор инструкций, ведущих себя по-разному в зависимости от того, в каком режиме они выполняются, в режиме ядра или в пользовательском режиме. В их число входят инструкции, осуществляющие ввод-вывод, изменяющие настройки блока управления памятью (MMU) и т. д. Попек и Голдберг назвали их служебными инструкциями (sensitive instructions), или инструкциями, чувствительными к виртуализации. Есть также набор инструкций, которые при выполнении в пользовательском режиме вызывают системные прерывания. Попек и Голдберг назвали их привилегированными инструкциями (privileged instructions). В статье этих специалистов впервые утверждалось, что машина может быть подвергнута виртуализации, только если служебные инструкции являются поднабором привилегированных инструкций. Проще говоря, при попытке сделать в пользовательском режиме то, что вы не должны делать в этом режиме, оборудование должно вызвать системное прерывание. В отличие от IBM/370, обладающей эти свойством, у Intel 386 его нет. При выполнении в пользовательском режиме будут проигнорированы или выполнены по-другому многие служебные инструкции 386-е машины.  
Эта проблема была окончательно решена, когда в начале 2005 года Intel и AMD представили виртуализацию на своих центральных процессорах (Uhlig, 2005). На центральных процессорах Intel она называлась технологией виртуализации (Virtualization Technology (VT)), а на центральных процессорах AMD она называлась безопасной виртуальной машиной (Secure Virtual Machine (SVM)). Далее в общем смысле будет использоваться термин VT.

Рассмотрим основные преимущества виртуализации:

* виртуальная машина работает под управлением гостевых операционных систем и содержит все стандартные компоненты компьютера, а значит виртуальная машина полностью совместима со стандартными операционными системами, программным обеспечением и т.д.;
* в рамках виртуальной машины можно работать с устаревшими программными решениями и операционными системами;
* возможность создать защищенные пользовательские окружения для работы с сетью, в этом случае вирусные атаки могут нанести вред операционной системе, а не виртуальной машине;
* несколько виртуальных машин, развернутых на физических ресурсах одного компьютера, изолированы друг от друга, таким образом, сбой одной из виртуальных машин не повлияет на доступность и работоспособность сервисов и приложений других;
* поскольку каждая виртуальная машина представляет собой программный контейнер, то она может быть перенесена или скопирована, как и любой иной файл;
* виртуальные машины не зависят от аппаратного обеспечения, на котором функционируют в том смысле, что в качестве значений параметров виртуальной машины, таких как оперативная память, процессор и т.п., можно указать значения и типы, отличающиеся от реальной физической конфигурации компьютера;
* виртуальные машины идеально подходят для процессов обучения и переподготовки, поскольку позволяют развернуть требуемую платформу вне зависимости от параметров и программного обеспечения хоста (физического компьютера, на котором функционирует виртуальная машина);
* возможность сохранения состояния виртуальной машины позволяет быстро вернуться к точке до внесения изменений в систему;
* в рамках одной гостевой операционной системы может быть развернуто несколько виртуальных машин, объединенных в сеть и взаимодействующих между собой;
* виртуальные машины могут создавать представления устройств, которых физически нет (эмуляция устройств).

Несмотря на то, что большая часть недостатков виртуальных машин разрешима, нельзя не упомянуть о них:

* обеспечение единовременной работы нескольких виртуальных машин потребует достаточного количества аппаратных мощностей;
* в зависимости от используемого решения, операционная система виртуальной машины может работать медленнее, чем на "чистом" аналогичном аппаратном обеспечении;
* различные платформы виртуализации не поддерживают виртуализацию всего аппаратного обеспечения и интерфейсов.

Рассмотрим основные типы виртуализации различных компонент.

* Виртуализация операционной системы.  
  Является наиболее распространенной в данный момент формой виртуализации. Виртуальная операционная система (виртуальная машина) представляет собой, как правило, совмещение нескольких операционных систем, функционирующих на одной аппаратной основе. Каждая из виртуальных машин управляется отдельно при помощи VMM (Virtual Machine Manager). Лидерами в области поставок решений для виртуализации информационных систем являются Microsoft, AMD, Intel и VMware.
* Виртуализация серверов приложений.  
  Под данным процессом виртуализации понимают процесс интеллектуальной балансировки нагрузки. Балансировщик нагрузки управляет несколькими веб - серверами и приложениями, как единой системой, пользователь, при этом, "видит" только один сервер, который, фактически, предоставляет функционал нескольких серверов.
* Виртуализация приложений.  
  Под виртуализацией приложений следует понимать использование программных решений в рамках изолированной виртуальной среды.
* Виртуализация сети.  
  Представляет собой объединение аппаратных и программных ресурсов в единую виртуальную сеть. Выделяют внутреннюю виртуализацию сети - создающую виртуальную сеть между виртуальными машинами одной системы, и внешнюю - объединяющую несколько сетей в одну виртуальную.
* Виртуализация аппаратного обеспечения.  
  В данном случае виртуализация заключается в разбиении компонент аппаратного обеспечения на сегменты, управляемые отдельно друг от друга. В некоторых случаях, виртуализация операционных систем невозможна без виртуализации аппаратного обеспечения.
* Виртуализация систем хранения.  
  В свою очередь делится на два типа: виртуализацию блоков и виртуализацию файлов.
* Виртуализация файлов, как правило используется в системах хранения, при этом ведутся записи о том, какие файлы и каталоги находятся на определенных носителях. Виртуализация файлов отделяет статичный указатель нахождения виртуального файла (C:\, к примеру) от его физического местоположения. Т.е. при запросе пользователем файла C:\file.doc решение виртуализации файлов отправит запрос к месту реального размещения файла.
* Виртуализация блоков. Используется в сетях распределенного хранения данных. Сервера - хранилища данных используют RAID - технологию. iSCSI интерфейс также использует блочную виртуализацию, позволяя операционной системе распределить виртуальное блочное устройство.
* Виртуализация сервисов.  
  По своей сути, виртуализация сервисов является объединением всех вышеуказанных типов виртуализации. Решение виртуализации сервисов позволяет работать с приложением вне зависимости от физического расположения его частей, объединяя и управляя их взаимодействием.

Приведенная выше типология рассматривает виртуализацию, в зависимости от части ИТ - инфраструктуры, в которой она применяется. Подходы к созданию интерфейсов между виртуальными машинами и системами виртуализации ресурсов также можно разделить на следующие типы:

* Полная виртуализация - технология, которая обеспечивает полную симуляцию базового оборудования, гостевая операционная система остается в нетронутом виде.
* Аппаратная виртуализация - технология, позволяющая запускать на одном компьютере (хосте) несколько экземпляров операционных систем (гостевых операционных систем). При этом гостевые ОС независимы друг от друга и от аппаратной платформы.  
  Аппаратная виртуализация представляет собой набор инструкций, облегчающих выполнение операций на аппаратном уровне, которое до этого могли выполняться только программно, при этом затрачиваются дополнительные программные ресурсы.
* Паравиртуализация - техника виртуализации, при которой гостевые операционные системы подготавливаются для исполнения в виртуализированной среде, для этих целей в ядро ОС вносят незначительные изменения. Для взаимодействия с гостевой операционной системой используется API - интерфейс.

**Облака**Существует множество облаков. Некоторые из них относятся к публичным и доступны любому, кто согласен платить за использование ресурсов, другие же являются закрытыми облаками организаций. Также разные облака предлагают разные услуги. Некоторые из них дают своим пользователям доступ к физическому оборудованию, но большинство виртуализируют свою среду.

Национальный институт стандартов и технологий (США), являясь источником, к которому всегда можно прибегнуть, перечислил пять основных характеристик:

1. Самообслуживание по требованию (On-demand self-service). Пользователи должны иметь возможность получать ресурсы автоматически, без человеческого участия.
2. Широкий доступ по сети (Broad network access). Все эти ресурсы должны быть доступны по сети посредством стандартных механизмов, чтобы ими могли воспользоваться гетерогенные устройства.
3. Объединение ресурсов в пул (Resource pooling). Компьютерные ресурсы, принадлежащие поставщику, должны быть объединены в пул для обслуживания нескольких пользователей с возможностью динамического назначения и освобождения ресурсов. Пользователи обычно не знают точного местонахождения «своих» ресурсов и даже того, в какой стране они расположены.
4. Быстродействующая эластичность (Rapid elasticity). Должна быть предоставлена возможность эластичного получения и освобождения ресурсов, может быть, даже в автоматическом режиме, чтобы происходило незамедлительное масштабирование в соответствии с потребностями пользователей.
5. Учтенные услуги (Measured service). Поставщик должен вести учет потребленных ресурсов тем способом, который соответствует типу заранее оговоренных облачных услуг.
6. Пользователи и группы (абстракции). Модели управления доступом. Пользователь как контейнер прав, наследование прав. Идентификация, аутентификация, авторизация. Алгоритмы/методики аутентификации (в т.ч. многофакторная)

UNIX-подобные операционные системы являются многопользовательскими. Пользователи и группы в которых они состоят используются для управления доступом к системным файлам, каталогам и периферии. По умолчанию linux предлагает относительно простые механизмы контроля доступа. Расширить их можно при помощи LDAP и ACL, но в этой теме мы рассмотрим стандартные средства контроля допуска.

**Пользователи и Группы в Linux**

**Пользователи:**Пользователь - это любой кто пользуется компьютером. Пользователю назначается имя, имя должно быть уникальным в системе (linux есть зарезервированные имена, такие как «root», «hal», и «adm»). Имя может состоять из букв английского алфавита, арабских чисел и символов «\_»(нижний пробел) «.»(точка).

Помимо системного имени, в систему может занестись и хранится полное имя (например Ф.И.О)(full name)  реального пользователя. Например, пользователю newuser в реальной жизни может соответствовать человек по имени John Smith. Эта информация позволит лучше контролировать и идентифицировать пользователей системному администратору, тем более, если пользователей в системе сотни или даже тысячи.

Под каждого пользователя, создается свой каталог (home directory). В этот каталог попадает пользователь, после того как он авторизировался и в нем храниться личные файлы и папки пользователя. Все каталоги пользователей собраны в одном месте, обычно это /home.

Также, пользователю назначается командная оболочка (командный интерпретатор, используемый в операционных системах семейства UNIX). Например: /bin/bash, /bin/zsh, /bin/sh и.т.д. У многих дистрибутивов linux, для пользователей, по умолчанию назначается командная оболочка bash.

Каждому пользователю назначается идентификационный номер (User ID). Сокращенно номер обозначается как UID, является уникальным идентификатором пользователя. Операционная система отслеживает пользователя именно по UID, а не по их имени.

Также, каждому пользователю назначается пароль для входа в систему. Пароль храниться в зашифрованном виде (encripted). Для создания и изменения пароля используется команда passwd. Системный администратор может сам назначить пароль или дать возможность пользователю ввести свой пароль при первой авторизации.

Каждый пользователь принадлежит минимум к одной или нескольким группам. (пользователи и группы в linux)

**Группы:**Для разграничения прав в linux, помимо пользователей, существуют группы. Так же как и пользователь, группа обладает правам доступа к тем или иным каталогам, файлам, периферии (в системе есть зарезервированные группы).  Для каждого файла определён не только пользователь, но и группа. Группы группируют пользователей для предоставления одинаковых полномочий на какие-либо действия.

Каждой группе назначается идентификационный номер (group ID). Сокращённо GID, является уникальный идентификатором группы. Принадлежность пользователя к группе устанавливается администратором.

***Ну и из другого сайта***Основа распределения прав доступа в операционной системе Linux лежит на понятии пользователь. Пользователю-владельцу файла выдаются определенные полномочия для работы с ним, а именно на чтение, запись и выполнение. Также отдельно устанавливаются полномочия на чтение, запись и выполнение для всех остальных пользователей. Поскольку в Linux все есть файлом, то такая система позволяет регулировать доступ к любому действию в этой операционной системе с помощью установки прав доступа на файлы. Но еще при создании Linux, разработчики поняли, что этого явно недостаточно.

Поэтому и были придуманы группы пользователей. Пользователи могут объединяться в группы, чтобы уже группам выдавать нужные полномочия на доступ к тем или иным файлам, а соответственно и действиям. В этой статье мы рассмотрим группы пользователей в Linux, рассмотрим зачем они нужны, как добавить пользователя в группу и управлять группами.

**Что такое группы?**Как я уже сказал группы в Linux появились еще в самом начале разработки этой операционной системы. Они были разработаны для того, чтобы расширить возможности управления правами. Разберем небольшой пример, возьмем организацию в которой есть только один компьютер, у нас есть администраторы и пользователи. У каждого человека свой аккаунт на нашем компьютере. Администраторы могут настраивать систему, пользователям же лучше не давать воли, чтобы что-то не сломали. Поэтому администраторы объединяются в группу admin, и ей дается доступ ко всему оборудованию, реально же ко всем файлам в каталоге dev, а пользователи, объеденные в группу users, и этой группе дается возможность читать и записывать файлы в общий каталог, с помощью которого они могут обмениваться результатами своей работы. Мы могли бы назначать права для каждого пользователя отдельно разрешая ему доступ к тому или иному файлу, но это слишком неудобно. Поэтому и были придуманы группы. Говорите, да ничего, можно назначить? Ну а теперь представьте что наши пользователи - это процессы. Вот тут вся краса групп выходит на передний план, группы используются не столько для обеспечения доступа для пользователей, сколько для управления правами программ, особенно их доступом к оборудованию. Для сервисов создаются отдельные группы и пользователь, от имени которого запущен он, сервис может состоять в нескольких группах, что обеспечивает ему доступ к определенным ресурсам.

Ну и так, для порядку: пользователь был придуман, так как возникла ситуация, что уже множество людей стало пользоваться одной вычислительной машиной, а информация на ней может храниться и конфиденциальная, и ещё чтоб настройки не ломались, потому придумали пользователя – у него своя папка для файлов и свои настройки

Ну и потом группы появились, когда всем надоело настраивать каждого пользователя

***Идентификация*** — это процесс опознания, определения объекта по его свойствам.   
  
***Аутентификация*** — это процесс подтверждения подлинности объекта. Подтверждение подлинности осуществляется с помощью какого-либо уникального значения или свойства объекта.  
  
***Авторизация*** — это процесс проверки прав на доступ и предоставление доступа — после того как объект/пользователь/компьютер/сервис был идентифицирован или аутентифицирован, он может запросить доступ к каким-либо ресурсам.

При «входе в компьютер» мы проходим все три процесса, описанные выше:

Идентификация — мы указываем свой логин для входа, при этом будет осуществлена проверка, существует ли такой пользователь. Если пользователь существует, то его необходимо аутентифицировать.

Аутентификация — мы указываем свой пароль, если пароль верный, то пользователь аутентифицирован и мы получаем возможность входа в систему. Если влезать более глубоко, то после проверки пароля, так же происходит механизм авторизации, имеет ли право аутентифицированный пользователь входить в систему. Например, на контроллере домена Windows Active Directory, Вы, по умолчанию, не сможете войти в систему локально, если пользователь не входит в группу Доменные Администраторы. То есть в данном примере пользователь будет идентифицирован и аутентифицирован (введены верные имя пользователя и пароль), но не будет авторизован (нет прав для локального входа на контроллер домена).

Авторизация — мы пытаемся открыть файл, если нам даны права на только чтение файла, то мы его сможем открыть и прочитать. Но записать какие-либо изменения в файл мы не сможем, так как мы не авторизированы на запись в файл

Каждая надежная (secured) компьютерная система должна требовать от всех пользователей во время входа проходить аутентификацию. Ведь если операционная система не может быть уверена в том, кем именно является пользователь, она не может знать, к каким файлам и другим ресурсам он может иметь доступ. Несмотря на то что тема аутентификации может показаться слишком тривиальной, она намного сложнее, чем можно было бы ожидать

Ну я даже не знаю, что написать

Самый распространённый вид аутентификации – логин + пароль, дёшево и сердито, но нужно следить за сложностью пароля и его сменой, а то как-то бесполезно

Ну расскажем, что пароли не хранятся в открытом виде, мало того, они даже в просто зашифрованном не хранятся. В LINUX при назначении пароля к нему приписывается соль (случайное число), после чего шифруется, в результате имеем запись <user> <salt> <hash>, таким образом сложнее получить пару логин-пароль описанным ниже способом

/\*Если взломщик сумел заполучить зашифрованный пароль, схема может быть подвергнута следующей атаке. Сначала взломщик составляет словарь возможных паролей, как это сделали Моррис и Томпсон. Они заранее зашифровываются с использованием известного алгоритма. Сколько времени займет этот процесс, не имеет значения, поскольку он происходит еще до попытки взлома. Теперь, вооружившись списком пар паролей и зашифрованных паролей, взломщик наносит удар. Он считывает публично доступный файл паролей и извлекает из него все зашифрованные пароли. Эти пароли сравниваются с зашифрованными паролями в его списке. При каждом совпадении становятся известными регистрационное имя и незашифрованный пароль. Простой сценарий, запускаемый в оболочке, может автоматизировать этот процесс, и он может быть выполнен за доли секунды. Путем обычного запуска сценария можно получить десятки паролей.\*/

**Одноразовые пароли**Вообще-то благодаря изобретенной Лесли Лэмпортом (Lamport, 1981) изящной схеме, обеспечивающей пользователю безопасный вход в систему по небезопасной сети с использованием одноразовых паролей, можно обойтись и без такого блокнота. Метод Лэмпорта может использовать пользователь, работающий на домашнем персональном компьютере, чтобы войти на сервер через Интернет, даже если взломщики могут отслеживать и копировать весь трафик в обоих направлениях. Более того, в файловых системах ни на сервере, ни на домашнем компьютере пользователя не нужно хранить никаких секретов. Иногда этот метод называют односторонней цепочкой хэширования (one-way hash chain). Алгоритм основан на односторонней функции, то есть на функции y = f(x), обладающей свойством, позволяющим при наличии x легко получить y. Но при наличии y получить методом вычислений x невозможно. Входные и выходные данные должны быть одной и той же длины, например 256 бит. Пользователь выбирает секретный пароль, который нужно запомнить. Он также выбирает целое число n, соответствующее количеству одноразовых паролей, которое алгоритм может сгенерировать. К примеру, рассмотрим n = 4, хотя на практике должно использоваться намного большее значение n. Если секретный пароль — s, то первый пароль получается путем запуска односторонней функции n раз: P1 = f(f(f(f(s)))) Второй пароль получается путем запуска односторонней функции (n – 1) раз: P2 = f(f(f(s))) Для получения третьего пароля функция f запускается дважды, а для получения четвертого пароля — один раз. В общем виде Pi –1 = f(Pi). Здесь главное — усвоить, что при наличии любого пароля из этой последовательности нетрудно вычислить принадлежащий ей же предыдущий пароль, но невозможно вычислить следующий. К примеру, имея P2, нетрудно найти P1, но невозможно найти P3. Сервер инициализируется числом P0, представляющим собой f(P1). Это значение сохраняется в записи файла паролей, связанной с регистрационным именем пользователя, вместе с целым числом 1, показывающим, что следующим будет востребован пароль P1. Когда пользователь хочет войти в систему в первый раз, он посылает свое регистрационное имя на сервер, который отвечает отправкой целого числа 1, находящегося в файле паролей. Пользовательская машина в ответ отправляет P1, который может быть вычислен локально из значения s, набираемого на месте. Затем сервер вычисляет f(P1) и сравнивает полученное значение со значением, сохраненным в файле паролей (P0). Если значения совпадают, вход разрешается, целое число увеличивается до 2, а значение P0 в файле паролей переписывается значением P1. При следующем входе в систему сервер посылает пользователю число 2, а пользовательская машина вычисляет пароль P2. Затем сервер вычисляет f(P2) и сравнивает полученное значение с записью в файле паролей. Если значения совпадают, вход раз- решается, целое число увеличивается до 3, а пароль P2 в файле паролей записывается поверх пароля P1. Свойство, позволяющее этой схеме работать, основано на том, что даже если взломщик может перехватить Pi , у него не будет возможности вычислить из него значение Pi + 1, он может вычислить лишь значение пароля Pi –1, который уже был использован и не представляет никакой ценности. Когда будут использованы все n паролей, сервер инициализируется заново новым секретным ключом.

**Схема аутентификации «оклик — отзыв»**Еще один вариант называется окликом — отзывом (chalelenge — response). При его применении пользователь при регистрации выбирает алгоритм, например x2 . Когда пользователь входит в систему, сервер посылает ему аргумент, скажем 7, а тот в ответ на это набирает число 49. Алгоритм может быть различным по утрам и в полуденное время, в разные дни недели и т. д. Если устройство пользователя обладает достаточной вычислительной мощностью, такой как, например, у персонального компьютера, КПК или сотового телефона, то может быть использована более сложная форма оклика — отзыв. Пользователь заранее выбирает секретный ключ k, который сначала вручную помещается на сервер. Копия хранится (под защитой) на пользовательском компьютере. При входе в систему сервер отправляет пользовательскому компьютеру случайное число r, из которого затем вычисляется значение f(r, k) (где f — это общеизвестная функция), которое от- правляется обратно. Затем сервер выполняет вычисление и проверяет, согласуется ли возвращенный ему результат с результатом собственного вычисления. Преимущество такой схемы перед обычным паролем заключается в том, что если взломщик отследит и запишет весь трафик в обоих направлениях, он не сможет получить ничего, что бы помогло ему в следующий раз. Разумеется, функция f должна быть достаточно сложной для того, чтобы даже при большом количестве наблюдений взломщик не смог вычислить значение k. Лучше всего выбрать криптографические хэш-функции, в которых аргументами служат обработанные функцией исключающего ИЛИ (XOR) значения r и k. Эти функции известны своей труднообратимостью.

**Аутентификация с использованием физического объекта**Тут должно быть более-менее понятно, магнитные ключи, обычные ключи, смарт-карты, специальные флешки – имеющий сей объект может войти (лучше ещё пароль спросить на всякий случай)

**Аутентификация с использованием биометрических данных**Тут, надеюсь, тоже всё понятно, тело человека вообще довольно уникальная штука, так что вырванные глаза и оторванные руки – значительное усложнение доступа для нарушителя

1. Системы централизованного управления настройками ОС. Цели, примеры (AD GPO, Puppet/Chef), архитектура. Управление обновлениями (патч-менеджмент)

Прошу прощения, но вот так

**Что такое система управления конфигурацией?**

Предположим, что у вас есть парк серверов, выполняющих различные задачи. Пока серверов мало и вы не растёте, вы легко настраиваете каждый сервер вручную. Устанавливаете ОС (может быть, автоматизированно), добавляете пользователей, устанавливаете софт, вводя команды в консоль, настраиваете сервисы, правите конфиги ваших любимых текстовых редакторов (nanorc, vimrc), выставляете на них одинаковые настройки DNS-сервера, устанавливаете агент системы мониторинга, настраиваете syslog для централизованного сбора логов… Словом, работы довольно много и она не особенно интересна.

Я искренне верю, что хороший админ — ленивый админ. Он не любит делать что-то несколько раз. Первая мысль — написать пару скриптов, в котором будет что-то наподобие:  
  
servers.sh

servers="server00 server01 server02 server03 server04"

**for** server **in** $servers ; **do**

scp /path/to/job/file/job.sh $server:/tmp/job.sh

ssh $server sh /tmp/job.sh

**done**

job.sh

**#!/bin/bash**

apt-get update

apt-get install nginx

service nginx start

Вроде всё стало легко и хорошо. Нужно что-то сделать — пишем новый скрипт, запускаем. Изменения приходят на все серверы последовательно. Если скрипт хорошо отлажен — всё станет хорошо. До поры.  
  
Теперь представьте, что серверов стало больше. Например, сотня. А изменение долгое — например, сборка чего-нибудь большого и страшного (например, ядра) из исходников. Скрипт будет выполняться сто лет, но это полбеды.  
  
Представьте, что вам нужно сделать это только на определенной группе из этой сотни серверов. А через два дня нужно сделать другую большую задачу на другом срезе серверов. Вам придётся каждый раз переписывать скрипты и много раз проверять, нет ли в них каких-нибудь ошибок, не вызовет ли это какой-нибудь проблемы при запуске.  
  
Самое плохое — это то, что в подобных скриптах вы описываете действия, которые необходимо выполнить для приведения системы в определенное состояние, а не само это состояние. Значит, если система изначально находилась не в том состоянии, что вы предполагали, то всё обязательно пойдет не так. Манифесты Puppet декларативно описывают необходимое состояние системы, а вычисление, как к нему прийти из текущего состояния — задача самой системы управления конфигурацией.

Если правильно использовать серверы и потратить некоторое время на первичную настройку системы управления конфигурацией, можно добиться такого состояния парка серверов, что вам не потребуется логиниться на них для выполнения работы. Все необходимые изменения будут приходить к ним автоматически.

**AD GPO**

Групповая политика — это набор правил или настроек, в соответствии с которыми производится настройка рабочей среды приёма/передачи (Windows, X-unix и другие операционные системы с поддержкой сети). Групповые политики создаются в домене и реплицируются в рамках домена. Объект групповой политики (англ. Group Policy Object, GPO) состоит из двух физически раздельных составляющих: контейнера групповой политики (англ. Group Policy Container, GPC) и шаблона групповой политики (англ. Group Policy Template, GPT). Эти два компонента содержат в себе все данные о параметрах рабочей среды, которая включается в состав объекта групповой политики. Продуманное применение объектов GPO к объектам каталога Active Directory позволяет создавать эффективную и легко управляемую компьютерную рабочую среду на базе ОС Windows. Политики применяются сверху вниз по иерархии каталога Active Directory.

По умолчанию в иерархии каталога Active Directory создаются две групповые политики: Default Domain Policy (политика домена по умолчанию) и Default Domain Controller’s Policy (политика контроллера домена по умолчанию). Первая из них назначается домену, а вторая — контейнеру, в состав которого входит контроллер домена. Если вы хотите создать свой собственный объект GPO, вы должны обладать необходимыми полномочиями. По умолчанию правом создания новых GPO обладают группы Enterprise Administrators (Администратор предприятия) и Domain Administrators (Администраторы домена).

**Что такое Puppet?**

Puppet — система управления конфигурацией. Архитектура — клиент-серверная, на сервере хранятся конфиги (в терминах puppet они называются манифесты), клиенты обращаются к серверу, получают их и применяют. Puppet написан на языке Ruby, сами манифесты пишутся на особом DSL, очень похожем на сам Ruby.

//как тесен мир, где-то читал, что можно и на самих рубях писать манифесты

Ну ясно, что можно использовать для разных целей, вроде как в манифестах описаны состояния объектов: содержимое файла, версия приложения и т.д. (смотря о чём речь), таким образом, можно каждый день применять что-то типа «все программы: последняя версия» на компах и он всё это применит. Теоретически работает как rake, то есть позволяет сделать что угодно, да ещё и применять на многих машинах сразу

На самом деле, сейчас полностью разбираться в этом не нужно, а больше.. ну хз, информации теоретической и правда мало, зато документации сколько хочешь

**Patch Management** – это процесс управления обновлениями программного обеспечения (ПО), без которого вряд ли обходится хоть одна современная компания, думающая о безопасности своей ИТ-инфраструктуры.

**Обновления или патчи** — это дополнительное программное средство, которое применяется для исправления обнаруженных дефектов в программном обеспечении или изменения его функционала.

Существуют 2 типа обновлений:

1. для операционных систем и серверного ПО, которые применяются для поддержки надлежащего уровня безопасности и устранения дыр в защите;
2. для прикладного ПО (например, Microsoft Office, Adobe Acrobat или клиентские части бизнес-приложений), которые необходимы для решения возникших проблем с часто используемыми или важными библиотеками и другими частями исходного кода.

**Методы управления обновлениями**Метод управления обновлениями является комбинацией подхода к тестированию обновлений и подхода к развёртыванию релизов с обновлениями. О них мы и расскажем далее.

Два самых распространенных подхода к тестированию обновлений — это:

1. тестирование на локальных виртуальных машинах;
2. тестирование в полноценной тестовой среде.

В общем, дальше идёт речь о том, что обновления надо поставить в тестовой среде, включающей почти все возможные отрасли предприятия, чтобы знать, что может сломаться от нового обновления, а потом уже ставить патчи повсеместно (ну на лекциях он так и говорил)

Ну и для этого уже и существуют всякие программные решения

В вопросе есть слово архитектура, я не уверен, что это значит

1. Основные технологии защиты в ОС. DEP, ASLR, UAC/sudo

Под механизмами защиты ОС будем понимать все средства и механизмы защиты данных, функционирующие в составе ОС. Операционные системы, в составе которых функционируют средства и механизмы защиты данных, часто называют защищенными системами.  
Под безопасностью ОС будем понимать такое состояние ОС, при котором невозможно случайное или преднамеренное нарушение функционирования ОС, а также нарушение безопасности находящихся под управлением ОС ресурсов системы.  
Основной проблемой обеспечения безопасности ОС является проблема создания механизмов контроля доступа к ресурсам системы. Процедура контроля доступа заключается в проверке соответствия запроса субъекта предоставленным ему правам доступа к ресурсам. Кроме того, ОС содержит вспомогательные средства защиты, такие как средства мониторинга, профилактического контроля и аудита.

**DEP**Предотвращение выполнения данных (англ. Dáta Execútion Prevéntion, DEP) — функция безопасности, встроенная в Linux, Mac OS X, Android и Windows, которая не позволяет приложению исполнять код из области памяти, помеченной как «только для данных». Она позволит предотвратить некоторые атаки, которые, например, сохраняют код в такой области с помощью переполнения буфера. Обнаружив, что программа использует системную память неправильно, средство DEP принудительно закрывает программу и выдает соответствующее сообщение.  
В отличие от брандмауэра или антивирусной программы средство DEP не препятствует установке потенциально опасных программ. Вместо этого выполняется наблюдение, чтобы программы использовали системную память безопасным образом.

DEP работает в 2-х режимах:

* аппаратном, для процессоров, которые могут помечать страницы как «не для исполнения кода».
* программном, для остальных процессоров.

При использовании аппаратной реализации DEP все области памяти процесса помечаются как области, не содержащие исполняемого кода, если иное не указано явным образом. DEP работает на уровне страниц виртуальной памяти и, как правило, отмечает какую-либо страницу памяти, изменяя один бит элемента таблицы страниц (PTE).

Программно реализованная DEP по умолчанию защищает только отдельные системные файлы и не зависит от поддержки процессором аппаратной реализации DEP.

Эта функция впервые появилась во 2-м пакете обновлений для Windows XP. В Android данная функция появилась начиная с версии Android 4.1 Jelly Bean.  
В операционных системах Windows XP с пакетом обновлений 2 (SP2), Windows Server 2003 с пакетом обновлений 1 (SP1) и Windows Vista функция DEP реализована как на программном, так и на аппаратном уровне.

Итог:  
Основным преимуществом функции DEP является возможность предотвращения запуска вредоносного кода из области данных. Как правило, содержимое стека и кучи по умолчанию не является исполняемым кодом. При аппаратной реализации функция DEP вызывает исключение при запуске кода из указанных местоположений. Функция DEP, реализованная на программном уровне, предотвращает использование вредоносными программами механизма обработки исключений, существующего в Windows.

**ASLR**ASLR (англ. address space layout randomization — «рандомизация размещения адресного пространства») — технология, применяемая в операционных системах, при использовании которой случайным образом изменяется расположение в адресном пространстве процесса важных структур данных, а именно образов исполняемого файла, подгружаемых библиотек, кучи и стека.

Технология ASLR создана для усложнения эксплуатации нескольких типов уязвимостей. Например, если при помощи переполнения буфера или другим методом атакующий получит возможность передать управление по произвольному адресу, ему нужно будет угадать, по какому именно адресу расположен стек, куча или другие структуры данных, в которые можно поместить шелл-код. Сходные проблемы возникнут и при атаке типа «возврат в библиотеку» (return-to-libc): атакующему не будет известен адрес, по которому загружена библиотека. В приведённых примерах от атакующего скрыты конкретные адреса, и, если не удастся угадать правильный адрес, приложение скорее всего аварийно завершится, тем самым лишив атакующего возможности повторной атаки и привлекая внимание системного администратора.

Впервые выпущена в Windows Vista.

Недостатки.  
Для использования ASLR исполняемые файлы требуется собирать со специальными флагами. В результате в коде не будут использоваться постоянные адреса, но при этом:

* увеличится размер кода исполняемых файлов;
* увеличится время загрузки в память каждого исполняемого файла;
* возникнет дополнительная несовместимость с ПО и библиотеками, разработанным под версии ОС без ASLR.
* Кроме того, защиту можно обойти через истощение памяти

**UAC/sudo**Контроль учётных записей пользователей (User Account Control) – механизм подтверждения опасных действий со стороны пользователя. Программное обеспечение не может обойти UAC в Windows 8.1 (для некоторых других версий операционных систем Windows существуют известные способы обхода) и выполнить вредоносное действие без подтверждения пользователем. Впервые была внедрена в Windows Vista. (sudo - в Unix системах, для запуска от имени админа)  
В случае, если программа запрашивает действие, требующее прав администратора, выполнение программы приостанавливается и система выдает запрос пользователю. Окно с запросом размещается на защищённом рабочем столе, чтобы предотвратить «нажатие» программой кнопки разрешения.   
Вот (неполный) список действий, вызывающих срабатывание Контроля учётных записей пользователей:

* Изменения в каталогах %SYSTEMROOT% и %PROGRAMFILES% — в частности, установка/удаление программы, драйверов и компонентов ActiveX; изменение меню «Пуск» для всех пользователей.
* Установка обновлений Windows, конфигурирование Windows Update.
* Перенастройка брандмауэра Windows.
* Перенастройка самого Контроля учётных записей пользователей.
* Добавление/удаление учётных записей.
* Перенастройка родительских запретов.
* Настройка планировщика задач.
* Восстановление системных файлов Windows из резервной копии.
* Любые действия в каталогах других пользователей.
* Изменение текущего времени (изменение часового пояса не вызывает срабатывание Контроля учётных записей пользователей).
* Вызов Редактора реестра.
* Установка некоторых программ

В операционных системах Windows 7 и Windows 8 алгоритм работы UAC был значительно улучшен — настройки теперь не ограничивается простым включением и отключением службы, как это было в Windows Vista. Пользователь может изменять степень «надоедливости» UAC в четырех градациях.

1. Начальный этап загрузки - BIOS/UEFI, MBR/GPT

**Начальная загрузка** — сложный и многошаговый процесс запуска компьютера. Загрузочная последовательность — это последовательность действий, которые должен выполнить компьютер для запуска операционной системы (точнее загрузчика), независимо от типа установленной OS.

После включения компьютера в его оперативной памяти нет операционной системы. Само по себе, без операционной системы, аппаратное обеспечение компьютера не может выполнять сложные действия, такие как, например, загрузку программы в память. Таким образом мы сталкиваемся с парадоксом, который кажется неразрешимым: для того, чтобы загрузить операционную систему в память, мы уже должны иметь операционную систему в памяти.  
Решением данного парадокса является использование специальной компьютерной программы, называемой начальным загрузчиком. Эта программа не обладает всей функциональностью операционной системы, но её достаточно для того, чтобы загрузить другую программу, которая будет загружать операционную систему. Часто используется многоуровневая загрузка, в которой несколько небольших программ вызывают друг друга до тех пор, пока одна из них не загрузит операционную систему.

В современных компьютерах процесс начальной загрузки начинается с выполнения процессором команд, расположенных в постоянной памяти (например на IBM PC — команд BIOS), начиная с предопределённого адреса (процессор делает это после перезагрузки без какой бы то ни было помощи). Данное программное обеспечение может обнаруживать устройства, подходящие для загрузки, и загружать со специального раздела выбранного устройства (чаще всего загрузочного сектора данного устройства) загрузчик ОС.

**Загрузочное устройство** — устройство, которое должно быть проинициализировано до загрузки операционной системы. К ним относятся устройства ввода (клавиатура, мышь), базовое устройство вывода (видеокарта и дисплей), и устройство, с которого будет произведена загрузка ОС — дисковод, жесткий диск, CD-ROM, флэш-диск, SCSI-устройство, сетевая карта (при загрузке по сети; например, при помощи PXE).

**BIOS** (Basic Input/Output System) - базовая система ввода-вывода определяет ход процесса загрузки компьютера. Лишь только после этого происходит загрузка операционной системы ПК и дальнейшая его работа происходит уже под управлением ОС. Во время работы компьютера BIOS обеспечивает базовые функции ввода-вывода информации и функции взаимодействия различных устройств между собой.

BIOS’у помогает **MBR**.  
Аббревиатура MBR означает Master Boot Record (Главная загрузочная запись). MBR представляет собой небольшой блок информации и размещается обычно в самом первом секторе жесткого диска или другого носителя информации. Как правило, объем MBR не превышает 512 байт.  
Назначение MBR двоякое. Во-первых, в MBR содержится исполняемый код, который позволяет BIOS начать загрузку необходимой операционной системы. Во-вторых, в MBR содержится информация о разделах жесткого диска – так называемая таблица разделов (Partition Table).  
Вызов кода, содержащегося в MBR, осуществляется в самом конце работы загрузчика BIOS, и фактически именно ему BIOS передает управление. Код, содержащийся в MBR, исследует и проверяет таблицу разделов, содержащуюся в загрузочном секторе, а затем передает управление загрузчику конкретного раздела, который содержит операционную систему.  
Концепция MBR впервые начала применяться еще в первых версиях DOS в начале 80-х гг. Несмотря на чрезвычайную простоту и относительную надежность данной технологии, она имеет и ряд недостатков. Основным недостатком MBR является отсутствие поддержки дисков объемом более 2 ТБ, которые на данный момент уже не являются редкостью. Также главная загрузочная запись не может поддерживать более 4 основных разделов на одном диске.  
Недостатки MBR побудили разработчиков искать пути их преодоления. Одной из альтернатив MBR стала технология GPT (GUID Partition Table). Этот новый формат таблицы разделов появился после внедрения современной технологии BIOS – UEFI.

**UEFI** (Unified Extensible Firmware Interface, Единый расширяемый интерфейс прошивки) разрабатывался компанией Intel как замена BIOS (Basic Input Output System). В отличие от 16-битного BIOS'а UEFI работает в 32- или 64-битном режиме, что позволяет использовать намного больше памяти для сложных процессов. Кроме того, UEFI приятно выглядит и там есть поддержка мышки.  
BIOS считывает информацию с первого сектора жесткого диска, где содержится главная загрузочная запись (MBR) и выбирает загрузочное устройство, где находится операционная система. Поскольку BIOS – это очень старая система, работающая с середины 1970-х гг, то она до сих пор работает в 16-битном режиме. Это обстоятельство ограничивает количество информации, которое может быть прочитано из системной ROM (постоянной памяти компьютера).  
UEFI выполняет ту же задачу, но делает это немного по-другому. Она хранит всю информацию об инициализации и начальной загрузке системы в специальном файле, находящемся на жестком диске в особом разделе ESP (EFI System Partition). Кроме того, ESP содержит загрузочные программы операционной системы, установленной на компьютере.

UEFI в перспективе предназначена для полной замены BIOS и предлагает много новых функций и улучшений, которые не могут быть реализованы в BIOS:

* Отсутствие ограничений объема разделов и их количества. Для хранения информации о жестком диске БИОС использует главную загрузочную запись (MBR), в то время как UEFI – так называемую GPT (GUID partition table). (см. разницу между ними)
* Скорость и производительность. Поскольку технология УЕФИ не зависит от конкретной платформы, то она способна уменьшить время загрузки и увеличить скорость работы компьютера, особенно в том случае, если в системе установлены объемные жесткие диски.
* Безопасность. Самым большим преимуществом UEFI по сравнению с БИОС является безопасность. Она реализуется при помощи технологии Secure Boot (безопасный запуск), поддерживаемой операционной системой Windows 8. Поскольку Secure Boot имеет возможность запрашивать цифровую подпись у загрузочных программ, то интерфейс UEFI позволяет использовать во время загрузки только аутентифицированные драйверы и службы. Secure Boot контролирует процесс загрузки до тех пор, пока операционная система полностью не загружена. Это дает гарантию того, что во время загрузки в компьютер не проникнет вредоносное ПО.
* Обратная совместимость. Большинство реализаций УЕФИ на компьютерах архитектуры PC поддерживают режим Legacy BIOS для дисков с MBR. Для этого в UEFI существует функция CSM (Compatibility Support Module, модуль поддержки совместимости). В случае диска с MBR загрузка производится в том же режиме, что и в системах на основе BIOS. Также возможна загрузка систем на основе BIOS с дисков, имеющих GPT.
* Поддержка сетевой загрузки. УЕФИ может осуществлять загрузку через сеть при помощи технологии Preboot eXecution Environment (PXE). Эта технология поддерживает основные сетевые протоколы, такие, как IPv4 и IPv6, UDP, DHCP и TFTP. Также поддерживается загрузка с загрузочных образов, хранящихся в сетевых хранилищах данных.
* Менеджер загрузки. В стандарте UEFI менеджер загрузки определяется как инструмент, предназначенный для загрузки операционной системы и всех необходимых драйверов. Загрузчики операционной системы хранятся в файлах, к которым может осуществляться доступ со стороны встроенного ПО. УЕФИ поддерживает файловые системы FAT32, а также FAT16 и FAT12 для съемных носителей. UEFI не зависит от загрузочных секторов, хотя ESP отводит для них место в целях обратной совместимости. Загрузчики автоматически определяются программным обеспечением UEFI, что позволяет осуществлять загрузку со съемных носителей.

**GPT** (GUID Partition Table, Таблица разделов GUID) — часть спецификации UEFI. UEFI использует GPT так же как BIOS использует MBR.  
GPT так же, как и MBR, располагается в начале загрузочного диска, однако занимает, как правило, не самый первый сектор, а второй сектор. Первый сектор резервируется для MBR, которая в дисках с GPT тоже может присутствовать, но лишь для совместимости со старыми операционными системами. Структура GPT похожа на структуру MBR, однако есть и отличия.

Главным отличием GPT от MBR, на мой взгляд, являются:

* Количество разделов: MBR поддерживает только 4 раздела. Можно и больше, но только через extended partition, что является просто хаком ограничений. GPT поддерживает до 128 разделов.
* Размер диска: MBR поддерживает диски до 2Тб, в то время как GPT — до 9.4 Зеттабайт (=9.4 × 10^21 байт, или условно 1000 Тб)
* Порядок загрузки: раньше BIOS загружал MBR, и в нем содержались адреса загрузчиков для каждого раздела диска. Теперь UEFI считывает GPT, находит в таблице все разделы типа efi (на них содержатся загрузчики), и подгружает их в память.

Кроме того, полезной особенностью GPT по сравнению с MBR является дублирование информации из таблицы разделов. Помимо оригинала GPT в начале диска существует ее копия в самом конце диска, позволяющая восставить таблицу разделов в случае повреждения.

Недостатком GPT, как, впрочем, и многих других новых технологий, является отсутствие ее поддержки во многих ОС. Многие ОС распознают и умеют читать GPT, но далеко не все умеют загружаться с ее помощью. Из 32-разрядных версий Windows GPT умеют читать все настольные операционные системы, начиная с Windows Vista, а также серверные ОС, начиная с Windows Server 2003 SP1. Из числа 64-разрядных систем GPT распознают все ОС, начиная с Windows XP Professional и Windows Server 2003.

1. Ситуации (сценарии) когда эффективнее использование процессов, когда - потоков

**Когда использовать потоки**

Когда вы стремитесь достичь повышенного параллелизма, упростить конструкцию и эффективнее использовать процессорное время.

**Повышенный параллелизм**Очень часто приложениям требуется выполнять несколько задач одновременно. Например, однажды я писал систему хранения документации для банка, которая получала данные с оптических дисков, хранящихся в устройствах с автоматической сменой оптических дисков. Представьте себе огромные массивы данных, о которых здесь идет речь, и устройство с одним приводом и 50 сменными дисками, обслуживающее до нескольких гигабайт данных. Иногда на загрузку диска и поиск запрошенного документа уходило не менее 5—10 секунд. Стоит ли говорить, что мое приложение не соответствовало бы идеалам производительности, если бы во время выполнения всех этих действий оно блокировало бы пользовательский ввод! Поэтому для обработки пользовательского ввода я создал другой поток, который выполнял физическую работу по получению данных и обеспечивал продолжение работы пользователя. Он уведомлял главный поток о загрузке документа. Это очень удачный пример разделения труда по загрузке документа и обработке пользовательского ввода между двумя потоками.

//здесь отделяем разные функции программы – работу с пользователем и работу с данными, то есть пользователь не должен ждать окончания чтения с диска, а может работать

**Упрощенная структура**Популярный способ упрощения структуры сложных систем — использование очередей и асинхронной обработки. Чтобы задействовать такую конструкцию, вам придется подготовить очереди для обработки событий, происходящих в вашей системе. Вместо прямого вызова методов создаются объекты и помещаются в очереди, в которых производится их обработка. На другом конце этих очередей работают многопоточные серверные программы, настроенные на отслеживание сообщений, приходящих в эти очереди. Преимущество упрощенных конструкций этого типа — надежность, устойчивость и расширяемость основанных на них систем.

//насколько я понял, все произошедшие события попадают в очередь, а на другом конце демоны ждут своего часа, чтобы всё это обработать, происходит такой псевдопараллелизм обработчиков

**Лучшее использование процессорного времени**

Часто ваше приложение реально не выполняет никакой работы, в то же время продолжая использовать свой квант. В моем примере с системой хранения документации один поток ожидал загрузки диска устройством. Очевидно, что это ожидание имело место из-за особенностей оборудования и не требовало процессорного времени. Другим аналогичным примером может служить ожидание вывода документов на печать или окончания операции ввода-вывода жесткого диска или CD-ROM. В каждом из этих случаев процессорное время не используется. Эти случаи являются кандидатами на перевод в потоки, работающие в фоновом режиме.

//по-сути обобщение первого пункта

**Когда лучше обойтись без потоков**

Распространенной ошибкой новичков является попытка развертывания нескольких потоков в каждом приложении. Это может быть намного хуже, чем вовсе не иметь потоков! Как и в случае любого другого инструмента из вашего арсенала средств программирования, потоки нужны только там, где без них не обойтись. Не применяйте несколько потоков в приложении как минимум в следующих случаях: когда затраты при этом превышают выгоды, когда вы не определили производительность для обоих вариантов и когда вы не можете сформулировать причину, по которой следует использовать потоки.

**Затраты больше выгод**

Написание многопоточных приложений занимает несколько больше времени и требует больших усилий. Бывает, что небольшие преимущества, которые дает использование нескольких потоков, просто не стоят времени, потраченного на реализацию многопоточности.

//ну просто любая ситуация, когда многопоточность не так уж важна, учитывая сложность

**Производительность для обоих случаев невозможно сравнить**

Если вы — новичок в программировании многопоточности, для вас может быть неожиданностью обнаружить, что издержки, связанные с созданием и диспетчеризацией потоков, могут привести к тому, что однопоточное приложение работает быстрее! Все зависит от того, что вы делаете и действительно ли вы можете распределить выполнение независимых задач между потоками. Например, если вам надо считать с диска три файла, создание для этого трех потоков не принесет вам пользы, поскольку все они будут обращаться к одному и тому же жесткому диску. Поэтому всегда старайтесь протестировать оба прототипа — однопоточную и многопоточную версию своей системы, прежде чем тратить время и средства на разработку решения, которое может быть неоправданно с точки зрения производительности.

//ну вот, даже пример более-менее понятный, мы кстати в работе для Жукова не три файла читали, а сразу много, да ещё и создавали другие и в них писали, вполне оправдано

**Нет веской причины**

Не следует рассматривать применение многопоточности как подход по умолчанию. Из-за врожденной сложности написания многопоточных приложений всегда следует ориентироваться на однопоточный код, если только у вас нет веской причины поступить иначе.

//ну и напоследок, не забываем про общее адресное пространство для потоков, они весьма удобно могут работать с одними и теми же данными, когда процессам нужны всякие трубы для общения

4. Классические задачи взаимодействия процессов: обедающие философы, читатели и писатели, спящий парикмахер

**«Проблема обедающих философов»**«Проблема обедающих философов» — классический пример, используемый в информатике для иллюстрации проблем синхронизации при разработке параллельных алгоритмов и техник решения этих проблем.

**Постановка задачи:**Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов.  
Каждый философ может либо есть, либо размышлять. Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Видимо, эти спагетти настолько скользкие, что философ может их есть только двумя вилками, причем взять он их может только справа и слева, т.е. нельзя тянуться через весь стол (альтернативная формулировка проблемы подразумевает миски с рисом и палочки для еды вместо тарелок со спагетти и вилок).  
Жизнь каждого их этих пяти философов состоит из чередования размышлений и приемов пищи. Каждый философ может взять ближайшую вилку (если она доступна), или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются раздельными действиями, которые должны выполняться одно за другим.  
Суть проблемы заключается в том, чтобы разработать модель поведения (параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышления.

**Самое просто и очевидно решение такое:**

void philosopher(int i) /\* i: номер философа (от 0 до 4) \*/

{

while (TRUE) {

think( ); /\* философ размышляет \*/

take\_fork(i); /\* берет левую вилку \*/

take\_fork((i+1) % N); /\* берет правую вилку; \*/ /\* ( % - оператор деления по модулю )\*/

eat(); /\* ест спагетти \*/

put\_fork(i); /\* кладет на стол левую вилку \*/

put\_fork((i+1) % N); /\* кладет на стол правую вилку \*/

}

}

К сожалению, этот алгоритм не является хорошим алгоритмом, т.к. может возникнуть ситуация, когда все философы берут левую вилку. Получается, что дальше они уже не могут взять правую вилку, а значит находятся в бесконечном её ожидании. Это состояние можно назвать состоянием взаимной блокировки.

Можно модифицировать задачу следующим образом: после того, как философ берет левую вилку, программа проверяет, доступна ли правая вилка. Если нет, философ должен положить левую вилку на место, а через некоторое время повторить попытку. Это исключит возможность взаимной блокировки, но по-прежнему остается возможность зацикливания. Например, при некоторой доле невезения, все философы одновременно берут левую вилку. Тогда через t1 минут они положат её, подождут ещё t2 минут, опять возьмут, и т.д. бесконечно. Т.е. наши философы опять будут голодать.  
Взаимное исключение является основной идеей «Проблемы обедающих философов». Эта проблема представляет собой общий, абстрактный сценарий, позволяющий объяснить проблемы этого типа. Ошибки философов наглядно демонстрируют те трудности, которые возникают в реальном программировании, когда нескольким программам требуется исключительный доступ к совместно используемым ресурсам. Эти вопросы изучаются в области параллельных вычислений.

**Решение задачи**

**Официант**Относительно простое решение задачи достигается путём добавления официанта возле стола. Философы должны дожидаться разрешения официанта перед тем, как взять вилку. Поскольку официант знает, сколько вилок используется в данный момент, он может принимать решения относительно распределения вилок и тем самым предотвратить взаимную блокировку философов. Если четыре вилки из пяти уже используются, то следующий философ, запросивший вилку, вынужден будет ждать разрешения официанта — которое не будет получено, пока вилка не будет освобождена. Предполагается, что философ всегда пытается сначала взять левую вилку, а потом — правую (или наоборот), что упрощает логику. Официант работает, как **семафор** — понятие, введённое Дейкстрой в 1965 году.  
Чтобы показать, как это решение работает, предположим, что философы обозначены от А до Д по часовой стрелке. Если философы А и В едят, то заняты четыре вилки. Философ Б сидит между А и В, так что ему недоступна ни одна из вилок. В то же время, философы Г и Д имеют доступ к одной неиспользуемой вилке между ними. Предположим, что философ Г хочет есть. Если он тут же берёт свободную вилку, то становится возможна взаимная блокировка философов. Если вместо этого он спрашивает разрешения у официанта, то тот просит его подождать — и можно быть уверенным в том, что как только пара вилок освободится, то по крайней мере один философ сможет взять две вилки. Таким образом, взаимная блокировка становится

**Иерархия ресурсов**Другое простое решение достигается путём присвоения частичного порядка ресурсам (в данном случае вилкам) и установления соглашения, что ресурсы запрашиваются в указанном порядке, а возвращаются в обратном порядке. Кроме того, не должно быть двух ресурсов, не связанных порядком, используемых одной рабочей единицей.  
Пусть ресурсы (вилки) будут пронумерованы от 1 до 5, и каждая рабочая единица (философ) всегда берёт сначала вилку с наименьшим номером, а потом вилку с наибольшим номером из двух доступных. Далее, философ кладёт сначала вилку с бо́льшим номером, потом — с меньшим. В этом случае, если четыре из пяти философов одновременно возьмут вилку с наименьшим номером, на столе останется вилка с наибольшим возможным номером. Таким образом, пятый философ не сможет взять ни одной вилки. Более того, только один философ будет иметь доступ к вилке с наибольшим номером, так что он сможет есть двумя вилками. Когда он закончит использовать вилки, он в первую очередь положит на стол вилку с бо́льшим номером, потом — с меньшим, тем самым позволив другому философу взять недостающую вилку и приступить к еде.  
Данное решение было предложено Дейкстрой.  
В то время, как иерархия ресурсов позволяет избежать взаимных блокировок, данное решение не всегда является практичным, в особенности когда список необходимых ресурсов неизвестен заранее. Например, если рабочая единица удерживает ресурс 3 и 5 и решает, что ей необходим ресурс 2, то она должна выпустить ресурс 5, затем 3, после этого завладеть ресурсом 2 и снова взять ресурс 3 и 5. Компьютерные программы, которые работают с большим количеством записей в базе данных, не смогут работать эффективно, если им потребуется выпускать все записи с верхними индексами прежде, чем завладеть новой записью. Это делает данный метод непрактичным.

**Решение на основе монитора**Пример ниже показывает решение, где вилки не представляются явно. Философы могут есть, если ни один из их соседей не ест. Аналогично системе, где философы, которые не могут взять вторую вилку, должны положить первую вилку до того, как они попробуют снова.

В отсутствии блокировок, связанных с вилками, философы должны обеспечивать то, что начало принятия пищи не основывается на старой информации о состоянии соседей. Например: Если философ B видит, что A не ест в данный момент времени, а потом поворачивается и смотрит на C, A мог начать есть, пока философ B смотрит на C. Используя одну взаимоисключающую блокировку, можно избежать этой проблемы. Эта блокировка не связана с вилками, но она связана с решением процедур, которые могут изменить состояние философов. Это обеспечивается монитором.  
Алгоритм монитора реализует схему «проверить, взять и положить» и совместно использует взаимоисключающую блокировку. Заметьте, что философы, желающие есть, не будут иметь вилок.  
Если монитор разрешает философу, желающему есть, действовать, то философ снова завладевает первой вилкой, прежде чем взять уже свободную вторую.  
По окончании текущего приёма пищи философ оповещает монитора о том, что обе вилки свободны.  
Стоит заметить, что этот алгоритм монитора не решает проблемы голодания. Например, философ B может бесконечно ждать своей очереди, если у философов A и C периоды приёма пищи всё время пересекаются. Чтобы гарантировать также, что ни один философ не будет голодать, можно отслеживать, сколько раз голодный философ не ел, когда его соседи положили вилки на стол. Если количество раз превысит некий предел, такой философ перейдёт в состояние Голодания и алгоритм монитора форсирует процедуру завладения вилками, выполняя условие недопущения голодания ни одного из соседей.  
Философ, не имеющий возможности взять вилки из-за того, что его сосед голодает, находится в режиме полезного ожидания окончания приёма пищи соседом его соседа. Эта дополнительная зависимость снижает параллелизм. Увеличение значения порога перехода в состояние Голодание уменьшает этот эффект.

**Читатели и писатели  
Задача о читателях-писателях** — одна из важнейших задач параллельного программирования. Формулируется она так.

Есть область памяти, позволяющая чтение и запись. Несколько потоков имеют к ней доступ, при этом одновременно могут читать сколько угодно потоков, но писать — только один. Как обеспечить такой режим доступа?

Можно обойтись обычным мьютексом, но это неоптимально — компьютерная память, как правило, устроена так, что несколько потоков могут свободно читать и писать её (единственная проблема — нет гарантии, что в процессе обработки переменная внезапно не изменится). У этой проблемы есть несколько вариантов, разные и решения. Кому отдавать приоритет — читателю или писателю — остаётся за программистом.

/\* Мьютексы — это один из вариантов семафорных механизмов для организации взаимного исключения.  
Мьютекс отличается от семафора тем, что только владеющий им поток может его освободить, т.е. перевести в отмеченное состояние.   
короче, семафор может пускать несколько потоков на доступ к чему-либо, а мьютекс только один, остальные, естественно, блокируются\*/

**Первая задача о читателях-писателях (приоритет читателя)**Задача формулируется так:  
Пока память открыта на чтение, давать читателям беспрепятственный доступ. Писатели могут ждать сколько угодно.

**Вторая задача о читателях-писателях (приоритет писателя)**Задача формулируется так:  
Как только появился хоть один писатель, читателей больше не пускать. При этом читатели могут простаивать.

**Третья задача о читателях-писателях (честное распределение ресурсов)**Не допускать простоев. Другими словами: независимо от действий других потоков, читатель или писатель должен пройти барьер за конечное время.

**Спящий парикмахер**В информатике **проблема** спящего парикмахера — классическая задача синхронизации и межпроцессного взаимодействия (interprocess communication) в многозадачной операционной системе. Проблема заключается в обеспечении того, чтобы парикмахер работал, когда есть клиенты и отдыхал, когда клиентов нет. (приписывается Эдсгеру Дейкстре (1965))  
Аналогия основана на гипотетической парикмахерской с одним парикмахером. У парикмахера есть одно рабочее место и приемная с несколькими стульями. Когда парикмахер заканчивает подстригать клиента, он отпускает клиента и затем идет в приёмную, чтобы посмотреть, есть ли там ожидающие клиенты. Если они есть, он приглашает одного из них и стрижет его. Если ждущих клиентов нет, он возвращается к своему креслу и спит в нем.  
Каждый приходящий клиент смотрит на то, что делает парикмахер. Если парикмахер спит, то клиент будит его и садится в кресло. Если парикмахер работает, то клиент идет в приёмную. Если в приёмной есть свободный стул, клиент садится и ждёт своей очереди. Если свободного стула нет, то клиент уходит. Основываясь на наивном анализе, вышеупомянутое описание по идее должно гарантировать, что парикмахерская функционирует правильно с парикмахером, стригущим любого пришедшего, пока есть клиенты, и затем спящим до появления следующего клиента. На практике же существует несколько конфликтных ситуаций, которые иллюстрируют общие проблемы планирования.

Все эти конфликтные ситуации связаны с тем фактом, что действия и парикмахера, и клиента (проверка приёмной, вход в парикмахерскую, занятие места в приёмной, и т. д.) занимают неизвестное количество времени и/или могут происходить одновременно. Например, клиент может войти и заметить, что парикмахер работает, тогда он идет в приёмную. Пока он идет, парикмахер заканчивает стрижку, которую он делает и идет, чтобы проверить приемную, причём делает это быстрее направляющегося туда клиента. Так как в приёмной пока ещё никого нет (клиент ещё не дошел), он возвращается к своему месту и спит. Парикмахер теперь ждет клиента, а клиент ждет парикмахера. В другом примере два клиента могут прибыть в то же самое время, когда в приемной есть единственное свободное место. Они замечают, что парикмахер работает, идут в приёмную, и оба пытаются занять единственный стул.

Существует несколько возможных **решений** данной проблемы. Основной элемент каждого из решений — мьютекс — механизм, который гарантирует, что изменить состояние (state) в данный момент времени может только один из участников. Парикмахер должен захватить мьютекс, прежде чем проверить клиентов, и освободить его, когда он начинает или спать, или работать. Клиент должен захватить тот же мьютекс, прежде чем войти в парикмахерскую, и освободить его, как только он займет место или в приемной, или у парикмахера. Это устраняет обе проблемы, упомянутые в предыдущей секции. Возможно также использование более общего механизма семафоров для указания текущего состояние системы. Например, при помощи семафора можно выразить число людей в приемной.

У варианта той же задачи с несколькими парикмахерами есть дополнительная сложность координирования нескольких парикмахеров среди ждущих клиентов.

Семафоры  
Семафор — это примитив синхронизации, позволяющий ограничить доступ к критической секции только для N потоков. При этом, как правило, семафор позволяет реализовать это без использования занятого ожидания. Концептуально семафор включает в себя счетчик и очередь ожидания для потоков. Интерфейс семафора состоит из двух основных операций: опустить (down) и поднять (up). Операция опустить атомарно проверяет, что счетчик больше 0 и уменьшает его. Если счетчик равен 0, поток блокируется и ставиться в очередь ожидания. Операция поднять увеличивает счетчик и посылает ожидающим потокам сигнал пробудиться, после чего один из этих потоков сможет повторить операцию опустить.

Мьютекс (mutex)  
Мьютекс — от словосочетания mutual exclusion, т.е. взаимное исключение — это примитив синхронизации, напоминающий бинарный семафор с дополнительным условием: разблокировать его должен тот же поток, который и заблокировал.

Монитор   
— это механизм синхронизации в объектно-ориентированном программировании, при использовании которого объект помечается как синхронизированный и компилятор добавляет к вызовам всех его методов (или только выделенных синхронизированных методов) блокировку с помощью мьютекса. При этом код, использующий этот объект, не должен заботиться о синхронизации. В этом смысле монитор является более высокоуровневой конструкцией, чем семафоры и мьютексы.  
Переменная условия (condition variable) — примитив синхронизации, позволяющий реализовать ожидание какого-то события и оповещение о нем.  
Большинство мониторов поддерживают внутри себя использование переменных условия. Это позволяет нескольким потоком заходить в монитор и передавать управление друг другу через эту переменную.

5.Гарвардская и фоннеймановские модели памяти

Фон-неймановская (принстонская) и гарвардская архитектуры

Эти архитектурные варианты были предложены в конце 1940-х годов специалистами соответственно Принстонского и Гарвардского университетов США для разрабатываемых ими моделей компьютеров.

В большинстве современных ВМ для хранения программ и данных используется общая память. Для обращения к этой памяти используется общая системная шина, по которой в процессор поступают и команды, и данные. Такая организация получила название архитектуры фон Неймана или Принстонской архитектуры, а ВМ с такой архитектурой называют машинами фон-неймановского или принстонского типа. В них области для хранения программ (Program Space – PS) и данных (Data Space – DS) образуют единое пространство и могут размещаться в любом месте общей памяти. При этом нет никаких признаков, указывающих на тип информации в ячейке памяти. Содержимое ячейки интерпретируется ЦП, и задача программиста – следить за тем, чтобы данные и программа обрабатывались по-разному.

Эта архитектура имеет ряд важных достоинств. Наличие общей памяти позволяет оперативно перераспределять ее объем для хранения отдельных массивов команд и данных в зависимости от решаемых задач. Таким образом, обеспечивается возможность более эффективного использования имеющегося объема основной памяти в каждом конкретном случае применения микропроцессора. Использование общей шины для передачи команд и данных значительно упрощает отладку, тестирование и текущий контроль функционирования системы, повышает ее надежность.

Гарвардская архитектура характеризуется физическим разделением памяти команд (программ) и памяти данных. В ее классическом варианте программы и данные хранятся в двух раздельных памятях, каждая из которых соединяется с процессором отдельной шиной, что позволяет полностью совмещать во времени выборку и исполнение команд, т.е. одновременно с чтением-записью данных при выполнении текущей команды производить выборку и декодирование следующей команды. ЭВМ, спроектированные в соответствии с концепцией разделения памяти на два вида, называют машинами гарвардского типа. Такое разделение позволяет повысить быстродействие и упростить схемотехническую реализацию микропроцессорной системы.

Недостатки Гарвардской архитектуры связаны с необходимостью использования большего числа шин, а также с фиксированным объемом памяти, выделенной для команд и данных, назначение которой не может оперативно перераспределяться в соответствии с требованиями решаемой задачи. Поэтому приходится использовать память большего объема, коэффициент использования которой при решении различных задач оказывается более низким, чем в системах с Принстонской архитектурой.

6. RAID-массивы

Акроним RAID (Reudant Array of Independed Disks) избыточный массив независимых дисков, впервые был использован в 1988 году исследователями из института Беркли Паттерсоном (Patterson), Гибсоном (Gibson) и Кацем (Katz). Они описали конфигурацию массива из нескольких недорогих дисков, обеспечивающих высокие показатели по отказоустойчивости и производительности.

Наиболее "слабой" в смысле отказоустойчивости частью компьютерных систем всегда являлись жесткие диски, поскольку они, чуть ли не единственные из составляющих компьютера, имеют механические части. Данные записанные на жесткий диск доступны только пока доступен жесткий диск, и вопрос заключается не в том, откажет ли этот жесткий диск когда-нибудь, а в том, когда он откажет.

RAID обеспечивает метод доступа к нескольким жестким дискам, как если бы имелся один большой диск (SLED - single large expensive disk), распределяя информацию и доступ к ней по нескольким дискам, обеспечивая снижение риска потери данных, в случае отказа одного из винчестеров, и увеличивая скорость доступа к ним.

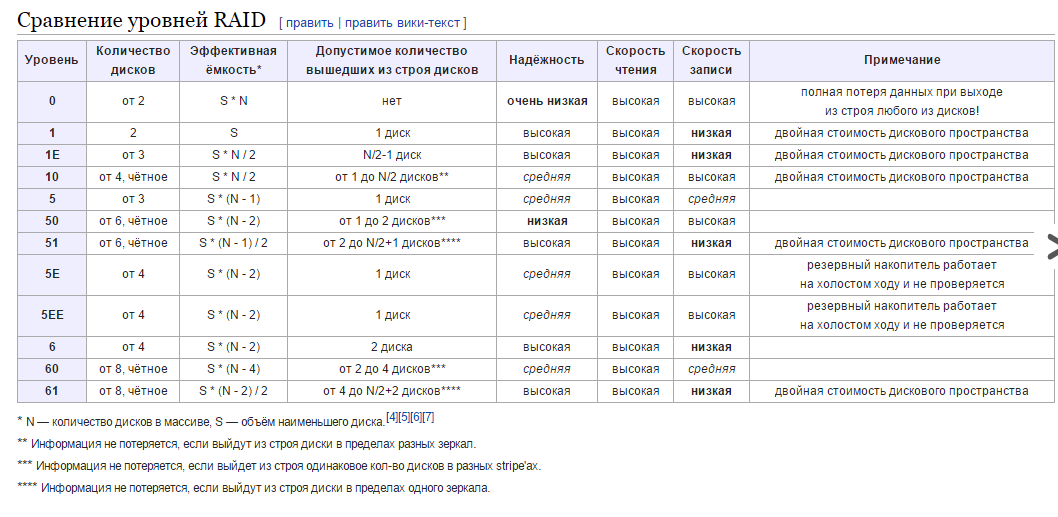
Обычно RAID используется в больших файл серверах или серверах приложений, когда важна высока скорость и надежность доступа к данным. Сегодня RAID находит применение так же в настольных системах, работающих с CAD, мультимедийными задачами и когда требуется обеспечить высокую производительность дисковой системы.

Калифорнийский университет в Беркли представил следующие уровни спецификации RAID, которые были приняты как стандарт де-факто: (цветом выделила те, которые мы рассмаривали на лекции. 2-4, как сказал Никонов, являются неудачными с точки зрения алгоритма)

* RAID 0 — дисковый массив повышенной производительности с чередованием, без отказоустойчивости;

(составляет из нескольких дисков «один», вместимостью равный сумме вместимости всех дисков)

* RAID 1 — зеркальный дисковый массив;
* RAID 2 зарезервирован для массивов, которые применяют код Хемминга;
* RAID 3 и 4 — дисковые массивы с чередованием и выделенным диском чётности;
* RAID 5 — дисковый массив с чередованием и «невыделенным диском чётности»;
* RAID 6 — дисковый массив с чередованием, использующий две контрольные суммы, вычисляемые двумя независимыми способами;
* RAID 10 — массив RAID 0, построенный из массивов RAID 1;
* RAID 50 — массив RAID 0, построенный из массивов RAID 5;
* RAID 60 — массив RAID 0, построенный из массивов RAID 6.



**RAID 0 (striping — «чередование»)**Режим, при использовании которого достигается максимальная производительность. Данные равномерно распределяются по дискам массива, дискиобъединяются в один, который может быть размечен на несколько. Распределенные операции чтения и записи позволяют значительно увеличить скорость работы, поскольку несколько дисков одновременно читают/записывают свою порцию данных. Пользователю доступен весь объем дисков, но это снижает надежность хранения данных, поскольку при отказе одного из дисков массив обычно разрушается и восстановить данные практически невозможно. Область применения - приложения, требующие высоких скоростей обмена с диском, например видеозахват, видеомонтаж. Рекомендуется использовать с высоконадежными дисками.  
(Никонов сказал, что «В чистом виде нигде особо не применяется»)

**RAID 1 (mirroring — «зеркалирование»)**Массив из двух дисков, являющихся полными копиями друг друга.(Из двух дисков видно один, второй – копия)  
Имеет высокую надёжность — работает до тех пор, пока функционирует хотя бы один диск в массиве. Вероятность выхода из строя сразу двух дисков равна произведению вероятностей отказа каждого диска, т.е. значительно ниже вероятности выхода из строя отдельного диска. На практике при выходе из строя одного из дисков следует срочно принимать меры — вновь восстанавливать избыточность. Для этого с любым уровнем RAID (кроме нулевого) рекомендуют использовать диски горячего резерва.  
Минусы: потеря вместимости в два раза.  
Плюсы: при потере одного диска, система всё равно работает. Так же может быть распараллелено чтение, за счет одинаковости данных на дисках.

**2,3,4** – неудачные с точки зрения алгоритмов. Основным недостатком уровней RAID от 2-го до 4-го является невозможность производить параллельные операции записи, так как для хранения информации о чётности используется отдельный контрольный диск.

**RAID 5**RAID 5 не имеет этого недостатка. Блоки данных и контрольные суммы циклически записываются на все диски массива, нет асимметричности конфигурации дисков. Под контрольными суммами подразумевается результат операции XOR (исключающее или). Xor обладает особенностью, которая даёт возможность заменить любой операнд результатом, и, применив алгоритм xor, получить в результате недостающий операнд. Например: a xor b = c (где a, b, c — три диска рейд-массива), в случае если a откажет, мы можем получить его, поставив на его место c и проведя xor между c и b: c xor b = a. Это применимо вне зависимости от количества операндов: a xor b xor c xor d = e. Если отказывает c тогда e встаёт на его место и проведя xor в результате получаем c: a xor b xor e xor d = c. Этот метод по сути обеспечивает отказоустойчивость 5 версии. Для хранения результата xor требуется всего 1 диск, размер которого равен размеру любого другого диска в raid.  
Емкость равна сумме N-1 дисков (при общем количестве N).  
Минимальное количество используемых дисков равно трём.

Достоинства  
RAID5 получил широкое распространение, в первую очередь, благодаря своей экономичности. Объём дискового массива RAID5 рассчитывается по формуле (n-1)\*hddsize, где n — число дисков в массиве, а hddsize — размер наименьшего диска. Например, для массива из четырех дисков по 80 гигабайт общий объём будет (4 — 1) \* 80 = 240 гигабайт. На запись информации на том RAID 5 тратятся дополнительные ресурсы и падает производительность, так как требуются дополнительные вычисления и операции записи, зато при чтении (по сравнению с отдельным винчестером) имеется выигрыш, потому что потоки данных с нескольких дисков массива могут обрабатываться параллельно.

Недостатки  
Производительность RAID 5 заметно ниже, в особенности на операциях типа Random Write (записи в произвольном порядке), при которых производительность падает на 10-25% от производительности RAID 0 (или RAID 10), так как требует большего количества операций с дисками (каждая операция записи, за исключением так называемых full-stripe write-ов, сервера заменяется на контроллере RAID на четыре — две операции чтения и две операции записи). Недостатки RAID 5 проявляются при выходе из строя одного из дисков — весь том переходит в критический режим (degrade), все операции записи и чтения сопровождаются дополнительными манипуляциями, резко падает производительность. При этом уровень надежности снижается до надежности RAID-0 с соответствующим количеством дисков (то есть в n раз ниже надежности одиночного диска). Если до полного восстановления массива произойдет выход из строя, или возникнет невосстановимая ошибка чтения хотя бы на еще одном диске, то массив разрушается, и данные на нем восстановлению обычными методами не подлежат. Следует также принять во внимание, что процесс RAID Reconstruction (восстановления данных RAID за счет избыточности) после выхода из строя диска вызывает интенсивную нагрузку чтения с дисков на протяжении многих часов непрерывно, что может спровоцировать выход какого-либо из оставшихся дисков из строя в этот наименее защищенный период работы RAID, а также выявить ранее не обнаруженные сбои чтения в массивах cold data (данных, к которым не обращаются при обычной работе массива, архивные и малоактивные данные), что повышает риск сбоя при восстановлении данных.

**RAID 6**RAID 6 — похож на RAID 5, но имеет более высокую степень надёжности — под контрольные суммы выделяется ёмкость 2-х дисков, рассчитываются 2 суммы по разным алгоритмам. Требует более мощный RAID-контроллер. Обеспечивает работоспособность после одновременного выхода из строя двух дисков — защита от кратного отказа. Обычно использование RAID-6 вызывает примерно 10-15% падение производительности дисковой группы, относительно RAID 5, что вызвано большим объёмом обработки для контроллера (необходимость рассчитывать вторую контрольную сумму, а также читать и перезаписывать больше дисковых блоков при записи каждого блока).  
Емкость равна сумме N-2 дисков (при общем количестве N).  
Для организации массива требуется минимум 4 диска.  
Встречается редко

**RAID 10 (1+0)**Эта архитектура представляет собой массив типа RAID 0, сегментами которого вместо отдельных дисков являются массивы RAID 1. Соответственно, массив этого уровня должен содержать как минимум 4 диска (и всегда чётное количество). RAID 10 объединяет в себе высокую отказоустойчивость и производительность.

Утверждение, что RAID 10 является самым надёжным вариантом для хранения данных вполне обосновано тем, что массив будет выведен из строя после выхода из строя всех накопителей в одном и том же массиве. При одном вышедшем из строя накопителе, шанс выхода из строя второго в одном и том же массиве равен 1/3\*100=33%. RAID 0+1 выйдет из строя при двух накопителях, вышедших из строя в разных массивах. Шанс выхода из строя накопителя в соседнем массиве равен 2/3\*100=66%, однако так как накопитель в массиве с уже вышедшим из строя накопителем уже не используется, то шанс того, что следующий накопитель выведет из строя массив целиком равен 2/2\*100=100%