Оглавление

[1. Что такое ОС? Взгляды "снизу" (управление ресурсами), "сверху" (абстракции), "со стороны" (эксплуатация, жизненный цикл). 2](#_Toc472596997)

[2. История развития ОС. 4](#_Toc472596998)

[3. Отечественные ОС. 10](#_Toc472596999)

[4. Свободные лицензии (BSD, GPL разных версий и др.). Свободный код не означает общедоступный или бесплатный. Совместимость, возможный переток кода, примеры. 13](#_Toc472597000)

[5. Коммерческие лицензии (OEM, BOX, аренда, подписка). Примеры. 15](#_Toc472597001)

[6. Классификация ОС. Не менее пяти критериев, пояснить выбор и содержание критериев. 17](#_Toc472597002)

[7. Процессы (абстракция). Состояния (жизненный цикл) процессов, создание, завершение. Реализация (таблица процессов). Модель многозадачности, оценка утилизации ресурсов процессора, накладные расходы. Процесс как контейнер ресурсов. 20](#_Toc472597003)

[8. Потоки (абстракция). Различные подходы к реализации, сравнение. Возможные проблемы при использовании потоков. 24](#_Toc472597004)

[9. Планирование при многозадачности, задачи. Критерии оценки эффективности, примеры. Вытесняющая и не вытесняющая многозадачности. Приоритеты, оптимальность, справедливость. Планирование потоков и процессов 29](#_Toc472597005)

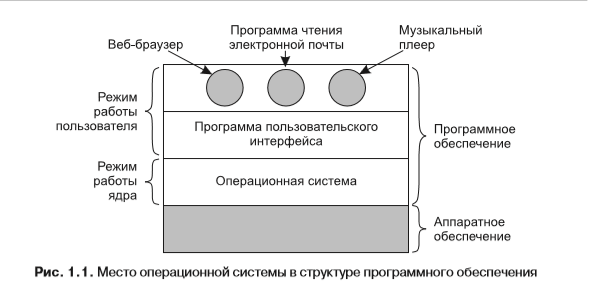
[10. Оперативная память. Адресация. Адресное пространство (абстракция). Подкачка. Виртуальная память, реализация (таблица). Алгоритмы заполнения и вытеснения страниц. 33](#_Toc472597006)

[11. Носители информации. Файлы и каталоги (абстракции). Структура файловой системы. Развитие атрибутов, прав доступа. Основные операции. Текущий каталог, переменные окружения. 34](#_Toc472597007)

[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. 40](#_Toc472597008)

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.**

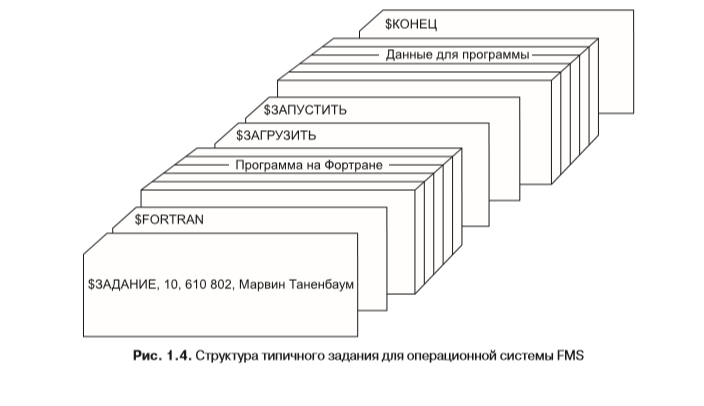
На заре компьютерной эры каждую машину проектировала, создавала, программировала, эксплуатировала и обслуживала одна и та же группа людей (как правило, инженеров). Все программирование велось исключительно на машинном языке или, и того хуже, за счет сборки электрических схем, а для управления основными 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.

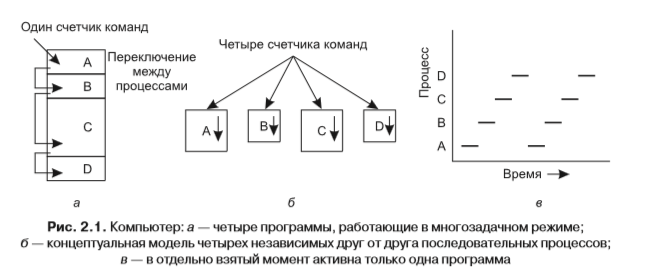
**Кластерные операционные системы**. Кластер – слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений и представляющихся пользователю единой системной, например, Windows 2000 Cluster Server, Windows 2008 Server, Sun Cluster (базовая ОС – Solaris).

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:

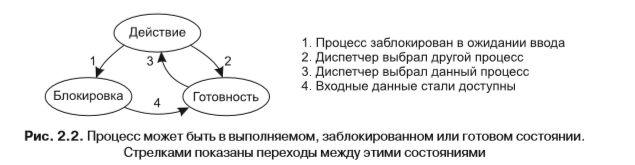
* Исполнения (у процесса есть и данные, и процессор (процессорное время)
* Готовности (данные есть, но процессора не хватает)
* Ожидания (не хватает данных, нет процессора)

Состояние ожидания контролируется ОС (как правило).

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

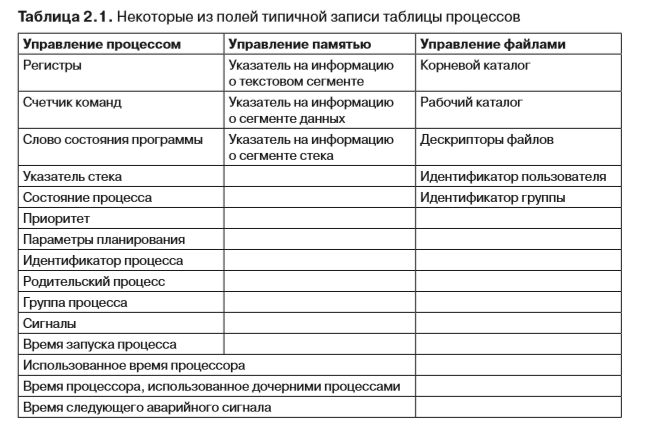
Ожидание->готовность->исполнение

)



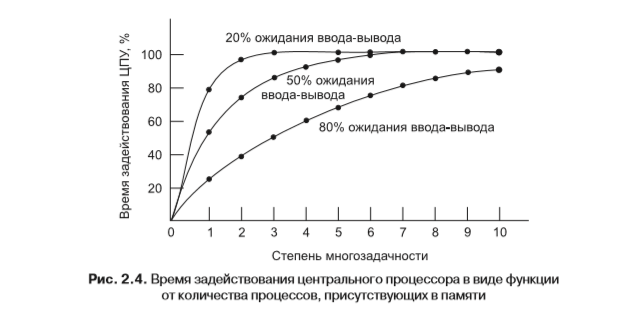
Переход 1 происходит в том случае, если операционная система определит, что процесс в данный момент выполняться не может. В некоторых системах для перехода в заблокированное состояние процесс может осуществить такой системный вызов, как pause.. Переходы 2 и 3 вызываются планировщиком процессов, который 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).

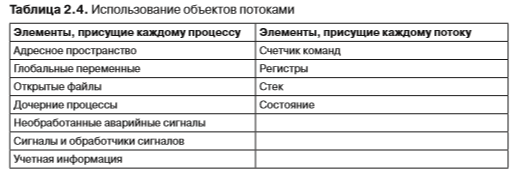
Справедливости ради следует заметить, что рассмотренная нами вероятностная модель носит весьма приблизительный характер. В ней безусловно предполагается, что все n процессов являются независимыми друг от друга, а значит, в системе с пятью процессами в памяти вполне допустимо иметь три выполняемых и два 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 %.

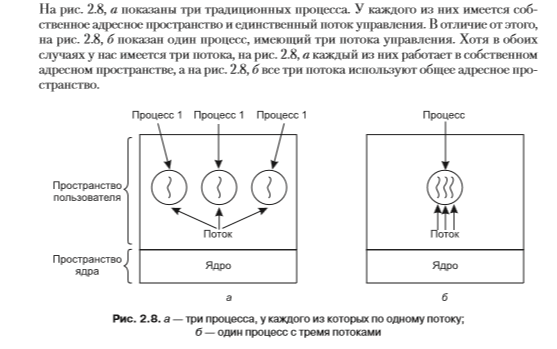
1. Потоки (абстракция). Различные подходы к реализации, сравнение. Возможные проблемы при использовании потоков.

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.

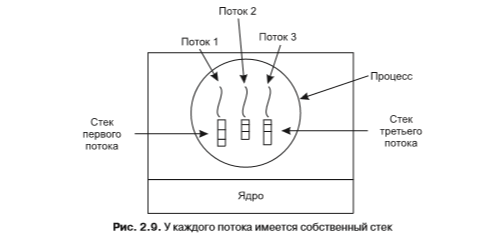
У потока есть счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять. У него есть регистры, в которых содержатся текущие рабочие переменные. У него есть стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры. Хотя 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.



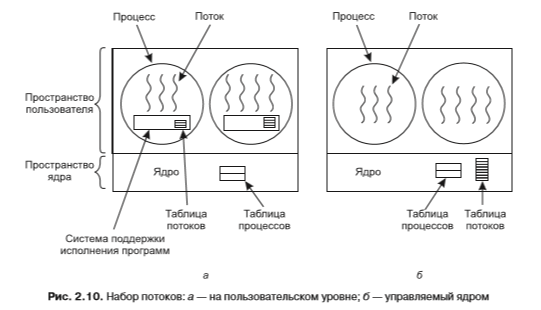
Нет необходимости (или даже возможности) указывать для нового потока какое-нибудь адресное пространство, поскольку он автоматически запускается в адресном пространстве создающего потока. Иногда потоки имеют иерархическую структуру, при которой у них устанавливаются взаимоотношения между родительскими 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.

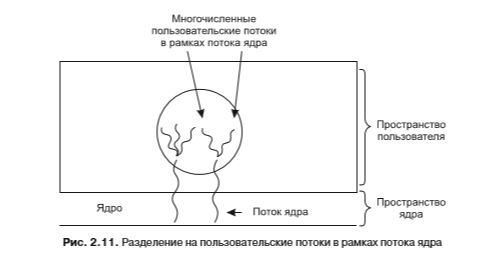
**Второй способ** – если ядро будет знать о потоках и управлять ими.  
В таблице потоков, находящейся в ядре, содержатся регистры каждого потока, состояние и другая информация. Вся информация аналогична той, которая использовалась для потоков, создаваемых на пользовательском уровне, но теперь она содержится 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:

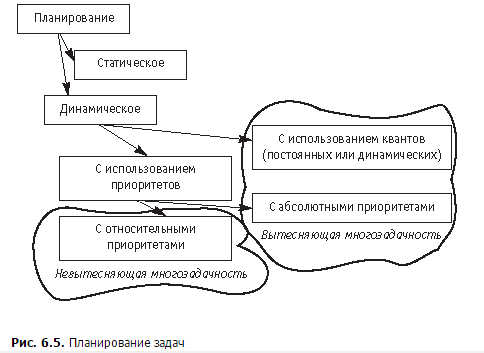
* Процессная многозадачность (основанная на процессах — одновременно выполняющихся программах). Здесь программа — наименьший элемент управляемого кода, которым может управлять планировщик операционной системы.
* Поточная многозадачность (основанная на потоках). Наименьший элемент управляемого кода —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

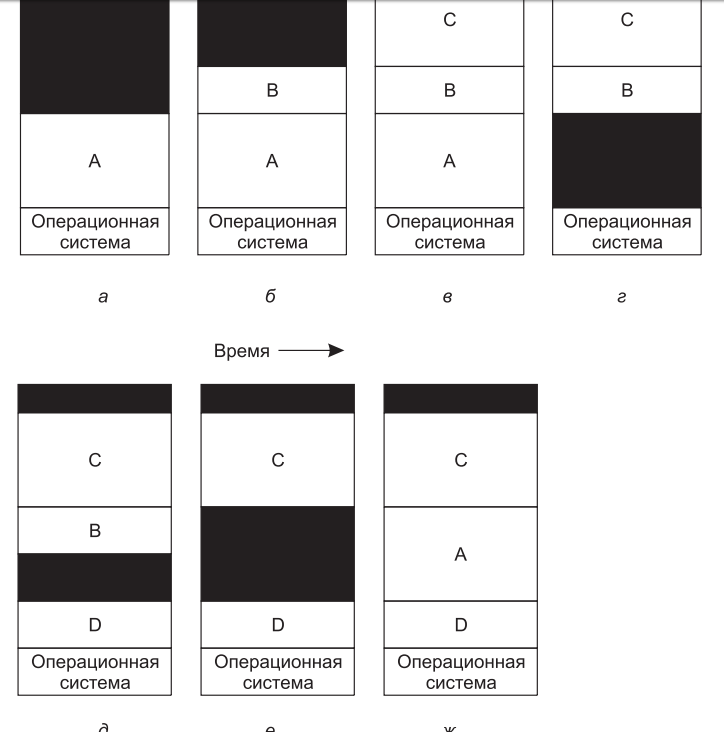
Адресное пространство не обязательно должно быть числовым. Набор интернет-доме- нов .com также является адресным пространством. Это адресное пространство состоит из всех строк длиной от 2 до 63 символов, которые могут быть составлены из букв, цифр и дефисов, за которыми следует название домена — .com. 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)

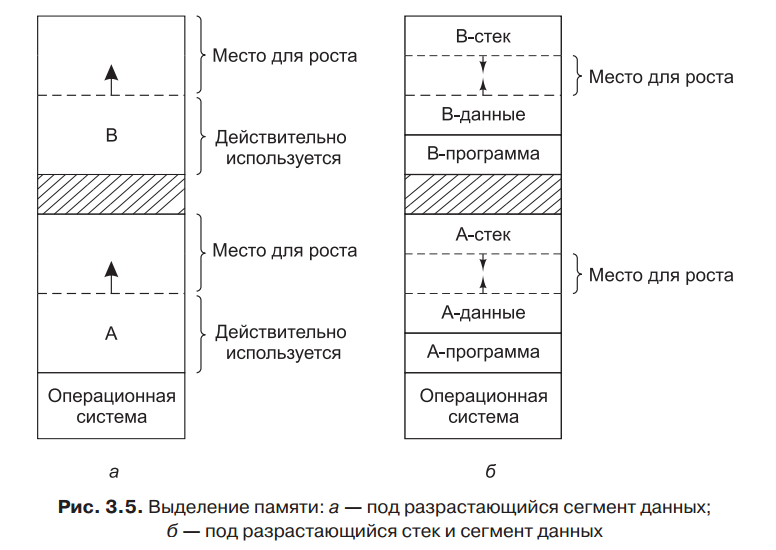


Типа понятная картинка, то что пропало либо завершилось, либо ушло полежать на диск

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

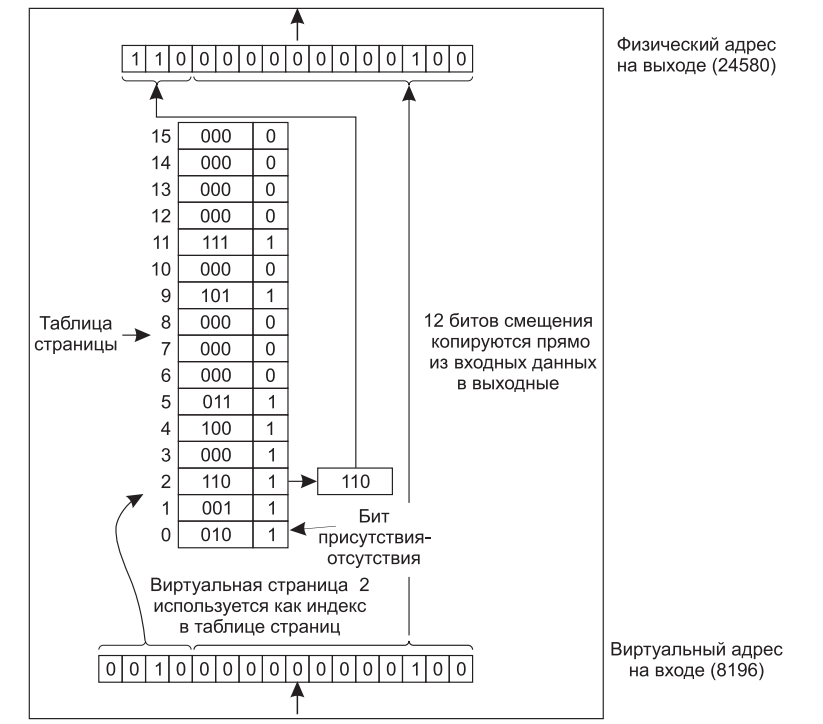
**Страничная организация памяти**В общем, для начала нам надо понять, что оперативной памяти у нас свободно 1Гб, а игре нужно 2, но она реально считает, что у неё их 2, так как создаётся виртуальная область памяти размером в 2Гб, при этом часть её в оперативной памяти, часть на диске (да, игра подвисает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

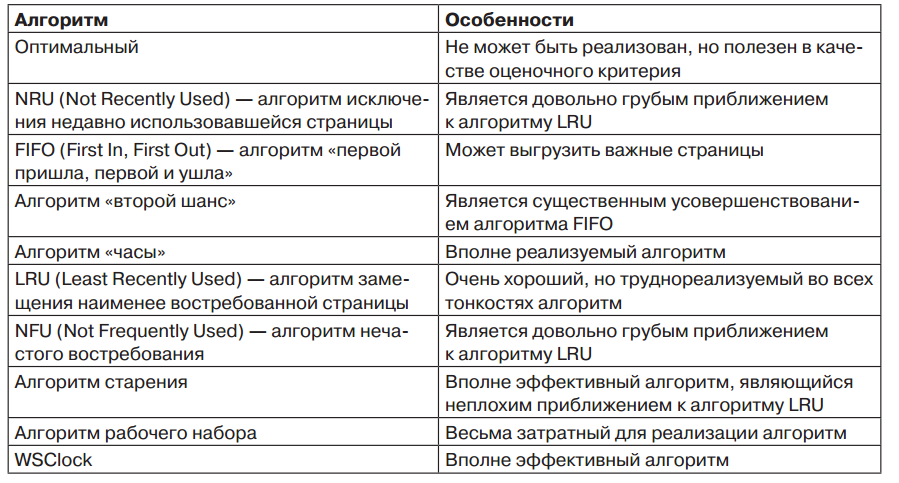
Базовый алгоритм рабочего набора слишком трудоемок, поскольку при возникновении ошибки отсутствия страницы для определения местонахождения подходящего кандидата на удаление необходимо просканировать всю таблицу страниц. Усовершенствованный алгоритм, основанный на алгоритме «часы», но также использующий 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).

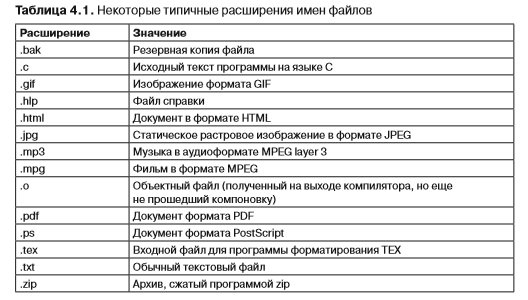
Если требуется быстрый доступ к информации, как, например, при выводе или передаче данных, то используются носители с вращающимся диском. Для архивирования, выполняемого периодически (Backup), наоборот, более предпочтительными являются ленточные носители. Они имеют большие объемы памяти в сочетании с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

3.По назначению носители информации различаются на три группы:

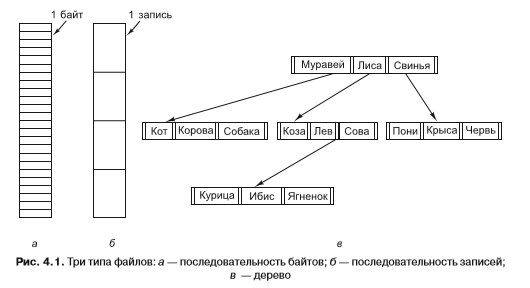
* распространение информации: носители с предварительно записанной информацией, такие как CD ROM или DVD-ROM;
* архивирование: носители для одноразовой записи информации, такие как CD-R или DVD-R (R (record able) – для записи);
* резервирование (Backup) или передача данных: носители с возможностью многоразовой 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

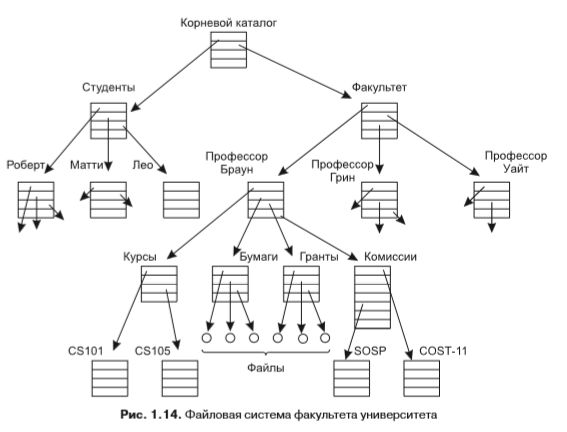


Многие операционные системы поддерживают несколько типов файлов. К примеру, в системах UNIX (опять же включая OS X) и Windows имеются обычные **файлы** **и каталоги**. В системе UNIX имеются также **символьные и блочные специальные файлы**. Обычными считаются файлы, содержащие информацию пользователя.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

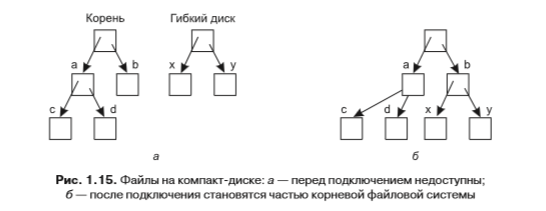


Иерархии файлов организованы в виде деревьев. Иерархии файлов обычно имеют глубину в четыре, пять и более уровней. Иерархия каталогов может существовать годами. Практически всегда существуют механизмы, позволяющие читать файлы и каталоги не только их владельцу, но и более широкой группе пользователей.

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.**

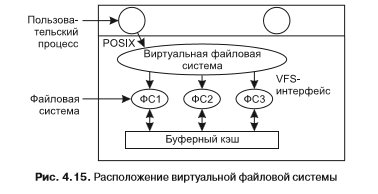
* Create (Создать каталог)
* Delete (Удалить каталог)
* Opendir (Открыть каталог)
* Closedir (Закрыть каталог)
* Readdir (Прочитать каталог). Этот вызов возвращает следующую запись из открытого каталога.
* Rename (Переименовать каталог).
* Link (Привязать). Привязка представляет собой технологию, позволяющую 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.

**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.