# Оглавление

[1. **Определение операционные системы. Системное и прикладное** 4](#_Toc124064450)

[**2.** **Функции** 7](#_Toc124064451)

[**3.** **Классификация ОС. Однозадачные и многозадачные ОС** 10](#_Toc124064452)

[**Классификация компьютерных систем** 10](#_Toc124064453)

[**Классификация операционных систем** 15](#_Toc124064454)

[**Эффективность и требования, предъявляемые к ОС** 18](#_Toc124064455)

[**Однозадачные и многозадачные** 21](#_Toc124064456)

[5. Классификация ОС. Микроядерные и макроядерные ОС 24](#_Toc124064457)

[**6.** **Классификация ОС. Специализированные операционные** 26](#_Toc124064458)

[**7.** **Классификация ОС. Операционные системы общего назначения** 29](#_Toc124064459)

[**8.** **Классификация ОС. Системы с асимметричной процессорной обработкой** 30](#_Toc124064460)

[**9.** **Классификация ОС. Системы с симметричной процессорной обработкой** 31](#_Toc124064461)

[**10.** **Процессы. Граф существования процесса (ТОЛЬКО ТУТ ПРО САМИ ПРОЦЕССЫ)** 32](#_Toc124064462)

[**11.** **Процессы. Классификация процессов** 35](#_Toc124064463)

[**12.** **Процессы. Программные потоки** 37](#_Toc124064464)

[**13.** **Управление ресурсами. Состояние процесса** 42](#_Toc124064465)

[**14.** **Сохранение состояния процесса** 43](#_Toc124064466)

[**15.** **Потоки. Средства синхронизации потоков** 44](#_Toc124064467)

[**16.** **Диспетчеризация процессора. Основная цель диспетчеризации процессора** 49](#_Toc124064468)

[**17.** **Планировщик. Разновидности стратегий, с точки зрения прерывания или избежания прерывания процессов, использует планировщик(ТУТ ПЛАНИРОВЩИК)** 51](#_Toc124064469)

[**18.** **Планировщик. Стратегия без прерывания процессов. Диаграмма Ганта(СХЕМЫ И ЕСТЬ ДИАГРАММА)** 64](#_Toc124064470)

[**19.** **Планировщик. Стратегия с прерыванием процессов. Диаграмма Ганта** 66](#_Toc124064471)

[**20.** **Планировщик. стратегии FCFS и каковы ее недостатки** 68](#_Toc124064472)

[**21.** **Планировщик. стратегии SJF (и SRTF) и оптимальность по какому критерию она обеспечивает** 70](#_Toc124064473)

[**22.** **Планировщик. Диспетчер. Диспетчеризации по приоритетам** 73](#_Toc124064474)

[**23.** **Планировщик. Проблема голодания процессов и каково ее решение в ОС** 75](#_Toc124064475)

[**24.** **Стратегии RR, оптимальность по какому критерию она обеспечивает и по какому критерию она хуже, чем SJF** 77](#_Toc124064476)

[Предсказание длины следующего периода активности 77](#_Toc124064477)

[Диспетчеризация по приоритетам 78](#_Toc124064478)

[Стратегия Round Robin (RR) 78](#_Toc124064479)

[Многоуровневая очередь 80](#_Toc124064480)

[**25.** **В чем заключается задача управления памятью? входная очередь заданий.** 82](#_Toc124064481)

[Устройство управления памятью 82](#_Toc124064482)

[**26.** **связывание адресов и на каких этапах обработки программы оно может выполняться** 84](#_Toc124064483)

[Связывание программ и данных с адресами в памяти 84](#_Toc124064484)

[**27.** **этапы обработки проходит программа на пути от исходного кода к двоичному образу в памяти** 85](#_Toc124064485)

[Многоэтапная обработка пользовательской программы 85](#_Toc124064486)

[Логическое и физическое адресное пространство 87](#_Toc124064487)

[**28.** **Адресация с использованием регистра перемещения** 88](#_Toc124064488)

[Устройство управления памятью 89](#_Toc124064489)

[**29.** **Оверлейная структура двухпросмотрового ассемблера** 91](#_Toc124064490)

[Оверлейная структура программы 91](#_Toc124064491)

[**30.** **откачка и подкачка. файл откачки. организовано смежное распределение памяти. Схема откачки и подкачки** 93](#_Toc124064492)

[Введение 93](#_Toc124064493)

[Откачка и подкачка 93](#_Toc124064494)

[Смежное распределение памяти 94](#_Toc124064495)

[Общая задача распределения памяти и стратегии ее решения 95](#_Toc124064496)

[Фрагментация 96](#_Toc124064497)

[**31.** **Схема адресации с аппаратной поддержкой регистров перемещения и границы** 98](#_Toc124064498)

[**32.** **Архитектура трансляции адресов при страничной организации** 100](#_Toc124064499)

[Страничная организация 100](#_Toc124064500)

[**33.** **таблицы страниц. Защита памяти** 102](#_Toc124064501)

[Реализация таблицы страниц 104](#_Toc124064502)

[Оценка среднего времени доступа к памяти с использованием TLB 106](#_Toc124064503)

[Защита памяти 106](#_Toc124064504)

[**34.** **Структура таблицы страниц. Хешированные таблицы страниц** 108](#_Toc124064505)

[Структура таблицы страниц 108](#_Toc124064506)

[Хешированные таблицы страниц 110](#_Toc124064507)

[**35.** **откачка и подкачка. файл откачки. Разделяемые страницы** 112](#_Toc124064508)

[Откачка и подкачка 112](#_Toc124064509)

[Разделяемые страницы 114](#_Toc124064510)

[**36.** **Принципы сегментной организации памяти. использования разделяемых сегментов** 116](#_Toc124064511)

[Введение 116](#_Toc124064512)

[Принципы сегментной организации памяти 116](#_Toc124064513)

[[Архитектура сегментной организации памяти 117](#_Toc124064514)

[Пример использования разделяемых сегментов 120](#_Toc124064515)

[**37.** **Схема адресации при сегментной организации памяти. Пример сегментной организации памяти** 122](#_Toc124064516)

[Архитектура сегментной организации памяти 122](#_Toc124064517)

[Пример использования разделяемых сегментов 125](#_Toc124064518)

[**38.** **Понятие файла. Структура файла. Методы доступа к файлам** 126](#_Toc124064519)

[**Введение** 126](#_Toc124064520)

[Общие сведения о файлах 128](#_Toc124064521)

[**Структура файла** 128](#_Toc124064522)

[Методы доступа к файлам 129](#_Toc124064523)

[**39.** **Сетевые топологии. Стратегии маршрутизации** 132](#_Toc124064524)

[Топология сетей 132](#_Toc124064525)

[Стратегии маршрутизации 136](#_Toc124064526)

[**40.** **Уровни сетевых протоколов (ISO). Структура сетевого сообщения, согласно модели ISO. Уровни протокола TCP/IP.** 138](#_Toc124064527)

[Введение 138](#_Toc124064528)

[Протоколы коммуникации 138](#_Toc124064529)

[Протокол TCP/IP 141](#_Toc124064530)

1. Определение операционные системы. Системное и прикладное программное обеспечение Место операционной системы в общей структуре компьютера

**Операционная система** ( ОС, в англоязычном варианте - operating system ) – базовое системное программное обеспечение, управляющее работой компьютера и являющееся посредником ( интерфейсом) между аппаратурой ( hardware ), прикладным программным обеспечением ( application software ) и пользователем компьютера ( user ). Фактически операционная система с точки зрения пользователя– это как бы продолжение аппаратуры, надстройка над ней, обеспечивающая более удобное,

надежное и безопасное использование компьютеров и компьютерных сетей.

Основные цели работы операционной системы следующие.

1. **Обеспечение удобства**, эффективности, надежности, безопасности выполнения пользовательских программ. Для пользователя самое главное – чтобы его программа работала,

вела себя предсказуемо, выдавала необходимые ему правильные результаты, не давала сбоев, не подвергалась внешним атакам. Вычислительную среду для такого выполнения программ и

обеспечивает операционная система.

2. **Обеспечение удобства**, эффективности, надежности, безопасности использования компьютера. Операционная система обеспечивает максимальную полезность и эффективность

использования компьютера и его ресурсов, обрабатывает прерывания, защищает компьютер от сбоев,

отказов и хакерских атак. Эта деятельность ОС может быть не столь заметной для пользователя, но она осуществляется постоянно.

3. **Обеспечение удобства**, эффективности, надежности, безопасности использования сетевых, дисковых и других внешних устройств, подключенных к компьютеру. Особая функция

операционной системы, без которой невозможно использовать компьютер, - это работа с внешними устройствами. Например, ОС обрабатывает любое обращение к жесткому диску, обеспечивая работу

соответствующего драйвера (низкоуровневой программы для обмена информацией с диском) и контроллера (специализированного процессора, выполняющего команды ввода-вывода с диском). Любая "флэшка", вставленная в USB-слот компьютера, распознается операционной системой, получает

свое логическое имя (в системе Windows – в виде буквы, например, G) и становится частью файловой системы компьютера на все время, пока она не будет извлечена (демонтирована).

4. Подчеркнем особую важность среди функций современных ОС обеспечения **безопасности**, **надежности и защиты данных**. Следует учитывать, что компьютер и операционная система работают в сетевом окружении, в котором постоянно возможны и фактически происходят атаки

хакеров и их программ, ставящие своей целью нарушение работы компьютера, "взлом" конфиденциальных данных пользователя, хранящихся на нем, похищение логинов, паролей, использование компьютера как "робота" для рассылки реклам или вирусов и др. В связи с этим в 2002

г. фирма Microsoft объявила инициативу по надежным и безопасным вычислениям (trustworthy computing initiative ), целью которой является повышение надежности и безопасности всего программного обеспечения, прежде всего – операционных систем. В данном курсе мы будем подробно останавливаться на том, какие действия по обеспечению надежности, безопасности и защите данных предпринимают современные ОС.

**Прикладное программное обеспечение** (applications software) – программы, предназначенные

для решения различных классов задач. К ним относятся, в частности, компиляторы, обеспечивающие

трансляцию программ с языков программирования, например, C++, в машинный код (команды);

системы управления базами данных (СУБД ); графические библиотеки, игровые

программы, офисные программы. Прикладное программное обеспечение образует следующий,

более высокий уровень, по сравнению с операционной системой, и позволяет решать на компьютере

различные прикладные и повседневные задачи.

Девизом фирмы Sun Microsystems еще в 1982 г., при ее создании, стал афоризм " The network is the

computer " (Сеть – это компьютер). Эту истину следует помнить всем пользователям компьютеров и их

операционных систем и шире использовать возможности компьютерных сетей, распределяя различные

функции между ее различными компьютерами (или хостами – hosts, как на компьютерном слэнге принято

называть компьютеры в сети). Изолированный от сети компьютер ныне – это "каменный век". Отсюда –

неразрывная связь операционных систем и сетей.

Общая картина функционирования компьютерной системы

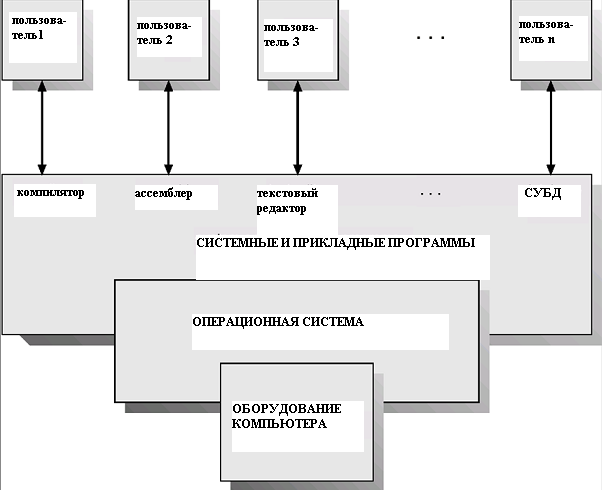


Рис. 1.1. Общая картина функционирования компьютерной системы

Пользователям компьютера доступны верхние уровни программного обеспечения – системные и прикладные программы (например, компиляторы, текстовые редакторы, системы управления базами данных). Эти программы взаимодействуют с операционной системой, которая, в свою очередь, управляет работой компьютера.

1. ФункцииОС, Операционные среды. Операционные оболочки

Сервисы (службы) ОС

Операционная система предоставляет для пользователей целый ряд сервисных возможностей, или,

коротко, сервисов (служб ):

**Исполнение программ** – загрузка программы в память и ее выполнение; например, в Windows при запуске программы ОС находит в файле ее двоичного кода ( .exe ) так называемую заглушку для **исполнения (execution stub)**, содержащую ссылку на код головного метода main , и запускает его. В

среде .NET этот же execution stub в файле двоичного кода используется системой для вызова не непосредственно исполняемой программы, а общего окружения времени выполнения – Common Language

Runtime (CLR), которое обеспечивает особый режим (managed execution) выполнения программы.

**Поддержка ввода-вывода** – обеспечение интерфейса для работы программ с устройствами вводавывода. Например, в UNIX у каждой программы есть свой стандартный ввод и стандартный вывод (по

умолчанию это терминал). В более старых ОС, например, IBM 360, привязку программы к устройствам

ввода-вывода требовалось специфицировать с помощью громоздких DD (Data Definition) – предложений на

специальном языке управления заданиями.

**Работа с файловой системой** – предоставление программам интерфейса для создания, именования, удаления файлов

**Коммуникация** – обмен информацией между процессами, выполняемыми на одном компьютере или на других системах, связанных в сеть. В операционных системах реализуется с помощью общей памяти (shared memory) или передачи сообщений.

**Обнаружение ошибок** в работе процессора, памяти, устройств ввода-вывода и программах пользователей.

**Дополнительные функции ОС**

Данные функции реализованы не непосредственно для удобства пользователя, а для обеспечения выполнения операций системы. Это следующие возможности.

**Распределение ресурсов** между пользователями, программами и процессами, работающими одновременно. Ведение статистики использования ресурсов, с целью выставления пользователям счетов (например, за сетевой трафик) или для анализа эффективности работы системы. **Защита** – обеспечение того, чтобы доступ к любым ресурсам был контролируемым.

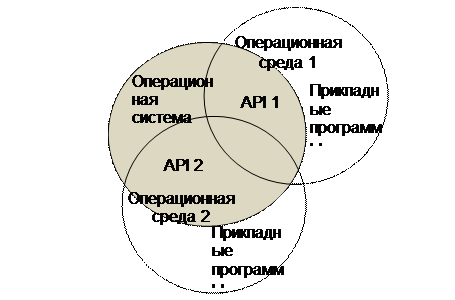
Программисту ОС дает так называемую операционную среду – набор системных функций (сервисов). В понятие операционная среда включают не только операционную систему, но и интерфейсы прикладных программ, прикладные программы, сетевые службы, базы данных и языки программирования. Таким образом, можно сказать, что термин «операционная среда» означает, прежде всего, соответствующие интерфейсы, необходимые программам и пользователям для обращения к управляющей части операционной системы с целью получить определенные сервисы.

Каждая операционная система имеет свое множество системных функций; они вызываются соответствующим образом, по принятым в системе правилам. Совокупность системных вызовов и правил, по которым их следует использовать создают интерфейс прикладного программирования (Application Program Interface, API) этой ОС.

Приложение выполняет обращение к функциям API с помощью системных запросов. Операционная система выполняет функции API, запуская специальные системные программные модули, входящие в ее состав. В высокоуровневых языках программирования обращение к системным функциям зачастую скрыто определенными синтаксическими конструкциями языка. Поэтому прикладной программист напрямую может к ним и не обращаться.

У разных операционных систем свой API. Очевидно, что программа, созданная в некоторой операционной системе с одним API, не будет работать в операционной системе с другим API.

Пытаясь преодолеть это ограничение, в ОС стали применять поддержку нескольких API. Таким образом, в общем случае операционная система может поддерживать несколько операционных сред. Итак, параллельное существование терминов «операционная система» и «операционная среда» вызвано тем, что операционная система (в общем случае) может поддерживать несколько операционных сред. Графически это можно представить на рис. 1.1.



Графическая иллюстрация понятий операционная система и операционная среда

**Графические оболочки ОС**

Как уже отмечалось, графическая оболочка имеется у каждой современной ОС. Графические оболочки ОСпредоставляют следующие основные возможности.

**Удобный графический пользовательский интерфейс**.

**Возможность выполнить любые системные настройки с помощью GUI**. Следует, по нашему личному опыту, особо отметить в этом отношении графические оболочки ОС Linux.

**Поддержка новых тенденций в развитии интерфейсов** – multi-touch, Tablet PC и др.

**Унификация графических оболочек для различных ОС**. В разных операционных системах используются графические оболочки CDE, KDE, GNOME. Возможно, в ближайшем будущем их список расширится.

Отметим, что оболочки для разных ОС практически неотличимы (все они имеют, например, кнопку Start, папку My Documents и т.д.), что удобно для конечных пользователей, так как упрощает изучение рабочей среды.

Однако, с нашей точки зрения, использование только графических оболочек (без изучения командных языков и конфигурационных файлов) снижает уровень подготовки системных программистов. При изучении ОС программистам, в особенности – системным, следует иметь это в виду. В этом отношении полезные навыки специалисту дает использование ОС типа UNIX – прежде всего, знание командных языков и конфигурационных файлов.

**Поддержка параллельных вычислений**

Все современные ОС предоставляют библиотеки для разработки параллельных алгоритмов с использованием возможностей аппаратуры. Они поддерживают следующие основные виды, стандарты и

инструменты параллелизма:

**OpenMP** - прагматы для распараллеливания программ для компиляторов с различных языков;

**MPI** (Message Passing Interface) – параллельное выполнение программ, взаимодействующих с помощью передачи сообщений.

ОС фирмы Microsoft поддерживают стандарт OpenMP. Имеются также библиотеки MPI.NET, поддерживающие стандарт MPI, а также библиотеки ParallelFx. Фирма Microsoft выпустила специальную версию операционной системы - Windows High Performance Computing (HPC) 2008, поддерживающую параллельное программирование

1. Классификация ОС. Однозадачные и многозадачные ОС **// чет хз(ЗДЕСЬ КЛАССИФИКАЦИЯ)**

## Классификация компьютерных систем

Для того, чтобы представить себе разнообразие и *масштабируемость* операционных систем, рассмотрим прежде всего классификацию современных компьютерных систем, для которых разрабатываются и используются ОС – от **суперкомпьютеров**до **мобильных устройств**, - и суммируем требования к ОС для этих классов компьютеров.

**Суперкомпьютеры (super-computers) –**мощные *многопроцессорные* компьютеры, наиболее современные из которых имеют *производительность* до нескольких **petaflops**(1015 вещественных операций в секунду; аббревиатура **flops**расшифровывается как **floating-point operations per second** ). Пример – *суперкомпьютер* "Ломоносов", установленный в МГУ. *Суперкомпьютеры* используются для вычислений, требующих больших вычислительных мощностей, сверхвысокой производительности и большого объема памяти. В реальной практике это прежде всего задачи моделирования – например, моделирования климата в регионе и прогнозирования на основе построенной модели погоды в данном регионе на ближайшие дни. Особенностью *суперкомпьютеров* является их параллельная *архитектура* – как правило, все они являются многопроцессорными. Соответственно, ОС для *суперкомпьютеров* должны поддерживать *распараллеливание* решения задач и синхронизацию *параллельных процессов*, одновременно решающих подзадачи некоторой программы.

**Многоцелевые компьютеры**, или **компьютеры общего назначения (mainframes)**– традиционное историческое название для компьютеров, распространенных в 1950-х – 1970-х гг., еще до эпохи всеобщего распространения персональных компьютеров. Именно для *mainframe*-компьютеров создавались первые ОС. Типичные примеры таких компьютеров: *IBM* 360/370; из отечественных – М-220, БЭСМ-6. На таких компьютерах решались все необходимые задачи – от расчета зарплаты сотрудников в организации до расчета траекторий космических ракет. Подобный *компьютер* выглядел достаточно неуклюже и громоздко и мог занимать *целый* большой зал. Вспомните, например. огромный *компьютер* *HAL* на космическом корабле в фантастическом фильме 1960-х гг. Стэнли Кубрика "Космическая одиссея 2001 г." Но никакие фантасты не смогли предвидеть прогресса компьютерной техники XXI века – прежде всего, того, что мощный *компьютер* будет не занимать целую комнату, а помещаться в небольшом ящике. Параметры ранних *mainframe*-компьютеров были весьма скромными: *быстродействие* - несколько тысяч операций в секунду, *оперативная память* – несколько тысяч ячеек (слов). Недостаточно удобным был пользовательский *интерфейс* (интерактивное взаимодействие с компьютерами было реализовано гораздо позже, в 1960-х гг.). Тем не менее, на таких компьютерах решались весьма серьезные задачи оборонного и космического назначения. С появлением персональных и портативных компьютеров классические *mainframe*-компьютеры ушли в прошлое. Однако следует подчеркнуть, что в именно в операционных системах для *mainframe*-компьютеров были реализованы все основные методы и алгоритмы, рассмотренные в данном курсе, которые впоследствии были использованы в ОС для персональных, карманных компьютеров и **мобильных устройств**.

**Кластеры компьютеров (computer clusters)**– группы компьютеров, физически расположенные рядом и соединенные друг с другом высокоскоростными шинами и линиями связи. Кластеры компьютеров используются для высокопроизводительных *параллельных вычислений*. Наиболее известны в мире компьютерные кластеры, расположенные в исследовательском центре *CERN* (Швейцария) – том самом, где находится большой адронный коллайдер. Как правило, компьютерные кластеры располагаются в исследовательских институтах и в университетах, в том числе, например, в Петродворцовом учебно-научном комплексе СПбГУ они используются в Петродворцовом телекоммуникационном центре (ПТЦ), на нашем математико-механическом и на физическом факультетах. *Операционная система* для кластеров должна, помимо общих возможностей, предоставлять средства для конфигурирования кластера, управления компьютерами (процессорами), входящими в него, распараллеливания решения задач между компьютерами кластера и мониторинга кластерной компьютерной системы. Примерами таких ОС являются ОС фирмы Microsoft – *Windows* 2003 for *clusters*; *Windows* 2008 *High-Performance Computing* (*HPC*).

**Настольные компьютеры (desktops)**– это наиболее распространенные в настоящее время компьютеры, которыми пользуются дома или на работе все люди, от школьников и студентов до домашних хозяек. Такой *компьютер* размещается на рабочем столе и состоит из монитора, системного блока, клавиатуры и мыши. Параметры современного (2010 г.) **настольного компьютера**, наиболее приемлемые для использования современных ОС: *быстродействие* процессора 1 – 3 ГГц, *оперативная память* – 1 – 8 *гигабайт* и более, объем жесткого диска (*hard* *disk* *drive* – *HDD*) – 200 Гб – 1 Тб и более (1 терабайт, Тб = 1024 Гб). Все разнообразие современных операционных систем (*Windows*, Linux и др.) – к услугам пользователей **настольных компьютеров**. При необходимости на **настольном компьютере**можно установить две или более операционных системы, разделив его дисковую *память* на несколько разделов (*partitions*) и установив на каждый из них свою операционную систему, так что при включении компьютера пользователю предоставляется стартовое *меню*, из которого он выбирает нужную операционную систему для загрузки.

**Портативные компьютеры (laptops, notebooks**– дословно "компьютеры, помещающиеся на коленях"; "компьютеры-тетрадки") – это миниатюрные компьютеры, *по* своим параметрам не уступающие настольным, но *по* своим размерам свободно помещающиеся в небольшую сумку или рюкзак или, например, на коленях пользователя, летящего в самолете в командировку и не желающего терять времени даром. Ноутбуки стоят обычно в несколько раз дороже, чем настольные компьютеры с аналогичными характеристиками. На **ноутбуках**используются те же операционные системы, что и для настольных компьютеров (например, *Windows* или MacOS). Характерными чертами **портативных компьютеров**являются всевозможные встроенные порты и адаптеры для беспроводной связи: *Wi-Fi* (официально *IEEE 802.11*) – вид радиосвязи, позволяющая работать в беспроводной сети с производительностью 10-100 *мегабит* в секунду (используется обычно на конференциях, в гостиницах, на вокзалах, аэропортах – т.е. в зоне радиусом в несколько сотен метров от источника приема-передачи); *Bluetooth* – также радиосвязь на более коротких расстояниях (10 – 100 м для *Bluetooth* 3.0), используемая для взаимодействия компьютера с мобильным телефоном, наушниками, плейером и др. **Внешние устройства**(дополнительные жесткие диски, принтеры, иногда даже *DVD-ROM*) подключаются к ноутбуку через порты *USB*. Еще лет 10 назад на ноутбуках активно использовались **инфракрасные порты**( **IrDA** ), которые, однако, неудобны, так как требуют присутствия "ответного" *IrDA* – порта другого устройства на расстоянии 20-30 см от порта ноутбука, при отсутствии между ними препятствий. Другая характерная черта ноутбуков – это наличие кард-ридеров – портов для чтения всевозможных карт памяти, используемых в *мобильных телефонах* или цифровых фотокамерах; обеспечивается также *интерфейс* FireWire (официально – *IEEE 1394*) для подключения цифровой видеокамеры; таким образом, ноутбуки хорошо приспособлены для ввода, обработки и воспроизведения обработки мультимедийной информации. Ныне портативный *компьютер* имеется почти у каждого студента, что они и используют для подготовки к ответу на экзамене, либо для решения задач практикума, иногда прямо в университетском буфете. Один из критических параметров ноутбука – время работы его батарей без подзарядки; очень хорошо, если это время составляет порядка 10 часов, что пока сравнительно редко; на компьютерах, используемых автором, это время составляет не более 5 часов. Популярная разновидность ноутбука ныне – это **нетбук**- ноутбук, предназначенный для работы в сети, обычно менее мощный и поэтому более дешевый, а также более миниатюрный.

**Карманные портативные компьютеры и органайзеры (КПК, handhelds, personal digital assistants – PDA)**– это "игрушки для взрослых" в виде миниатюрного компьютера, помещающегося на ладони или в кармане, но *по* своему быстродействию иногда не уступающего ноутбуку. При всей привлекательности, серьезные недостатки КПК, с точки зрения автора, - это неудобство ввода информации (приходится пользоваться палочкой- **стилусом**, - ведь не носить же с собой еще и громоздкую клавиатуру, - либо микроскопической выдвижной клавиатурой, на которой фактически тем же стилусом только и можно работать), а также неудобство чтения информации на маленьком экране. *Автор* уже "наигрался" в подобные КПК, - например, типа PalmPilot, - предпочитает пользоваться ноутбуками, а самым надежным органайзером считает ... небольшой бумажный блокнот. Однако молодежь приглашается к этой увлекательной интересной игре – через все в жизни нужно пройти. Современные КПК имеют фактически те же порты и адаптеры, что и ноутбуки – *Wi-Fi*, *Bluetooth*, *IrDA*, *USB*. Операционные системы для КПК аналогичны ОС для ноутбуков, но все же учитывают более жесткие ограничения КПК *по* объему оперативной памяти. В настоящее время для КПК широко используется ОС *Windows* *Mobile* – аналог *Windows* для *мобильных устройств*. До недавнего времени была также широко распространена PalmOS для органайзеров типа PalmPilot фирмы *3COM*. Разумеется, для КПК имеется аппаратура и *программное обеспечение* для подключения к ноутбуку или настольному компьютеру с целью синхронизации данных, что обеспечивает дополнительную *надежность*.

**Мобильные устройства (mobile intelligent devices – мобильные телефоны, коммуникаторы)**– это устройства, которыми каждый из нас пользуется постоянно для голосовой связи, реже – для записи или обработки какой-либо информации или для выхода в *Интернет*. С точки зрения автора, наиболее важные параметры мобильного устройства – это *по*-прежнему качество голосовой связи и время автономной работы батареи. Однако все большее *значение* приобретают встроенные в них цифровые фото- и видеокамеры. Операционные системы для *мобильных устройств* отличаются большей компактностью, ввиду более жестких ограничений *по* памяти (например, собственная *память*, не расширенная SmartMedia-картой, у используемого автором с 2006 г. мобильного телефона Nokia 3230, кстати, с неплохой встроенной фотокамерой – всего 30 *мегабайт*). Эпоха доминирования на рынке мобильных телефонов операционных систем типа Symbian, видимо, заканчивается, и они уступают *место* более современным и обеспечивающим лучший пользовательский *интерфейс* ОС *Google* *Android* и Microsoft *Windows* *Mobile*. Для *мобильных устройств*, как и для КПК, весьма важная характеристика ОС – это ее *надежность*, в частности, *сохранность данных* после переполнения памяти, возникающего, например, в результате приема большого числа *SMS*-сообщений, интенсивной фото- или видеосъемки. К сожалению, ОС Symbian, *по* опыту автора, ведет себя в этом отношении недостаточно надежно: при приеме "лишнего" *SMS*-сообщения, не помещающегося в *память* телефона, он ... блокируется с жалобным сообщением "*Memory* *full*", после чего приходится вручную отключать и включать телефон и таким образом перезагружать ОС (к сожалению, *SMS*-ка, переполнившая чашу терпения ОС, при этом пропадает). С этой точки зрения, ОС для *мобильных устройств* нуждаются в дальнейшем совершенствовании. Что касается прочего программного обеспечения для мобильных телефонов (игр, утилит, прикладных программ и др.), то при их разработке доминирует *Java*-технология – одна из постоянных областей научных интересов автора. Изучить основы *Java* Вы можете, например, *по* книге автора "Введение в *Java*-технологию" или *по* его *Интернет*-курсу, опубликованному на русском сайте *Sun* / *Oracle*. В *мобильных телефонах* используется платформа ("издание") JME – **Java Micro Edition**, и любой *мобильный телефон*, выпускаемый вот уже более 10 лет, поддерживает *Java*. Программы на *Java* для мобильных телефонов называются **мидлетами**(от аббревиатуры **MID – Mobile Intelligent Device** ).

**Носимые компьютеры (wearable computers) –**для повседневной жизни достаточно экзотические устройства, однако для специальных применений (например, встроенные в скафандр космонавта или в кардиостимулятор) они жизненно важны. Разумеется, их *память* и *быстродействие* значительно меньше, чем у настольных компьютеров, но критическим фактором является их сверхвысокая *надежность*, а для их операционных систем и прочего программного обеспечения – минимальное возможное **время ответа (response time)**– *интервал*, в течение которого система обрабатывает информацию от датчиков, от пользователя или из сети, превышение которого грозит катастрофическими последствиями. С этой точки зрения, ОС для **носимых компьютеров**можно отнести к **системам реального времени**.

**Распределенные системы (distributed systems)**– это системы, состоящие из нескольких компьютеров, объединенных в проводную или беспроводную *сеть*. Фактически, таковы ныне все компьютерные системы (вспомните девиз **"Сеть – это компьютер** "). Все операционные системы должны, таким образом, поддерживать распределенный режим работы, средства сетевого взаимодействие, высокоскоростную надежную передачу информации через *сеть*. Все эти вопросы подробно рассмотрены в данном курсе.

**Системы реального времени (real-time systems) –**вычислительные системы, предназначенные для управления различными техническими, военными и другими объектами в режиме реального времени. Характеризуются основным требованием к аппаратуре и программному обеспечению, в том числе к операционной системе: **недопустимость превышения времени ответа**системы, т.е. ожидаемого времени выполнения типичной *операции* системы. Для ОС требования реального времени накладывают весьма жесткие ограничения – например, в основном цикле работы системы недопустимы прерывания (так как они приводят к недопустимым временным затратам на их обработку). *Системы реального времени* – особая весьма серьезная и специфическая область, изучение которой выходит за рамки данного курса.

Приведенный обзор дает некоторое *представление* о разнообразии компьютерных систем в наше время. Для каждой из них должна быть разработана адекватная *операционная система*.

## Классификация операционных систем

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 [[10](https://intuit.ru/studies/courses/631/487/literature#literature.10)].

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.

1. 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.).
2. 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).

1. *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.

1. *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).

## Эффективность и требования, предъявляемые к ОС

К операционным системам современных компьютеров предъявляется ряд требований. Главным требованием является выполнение основных функций эффективного управления ресурсами и обеспечения удобного интерфейса для пользователя и прикладных программ. Современная ОС должна поддерживать мультипрограммную обработку, виртуальную *память*, *свопинг*, развитый *интерфейс* пользователя (многооконный графический, аудио -, менюориентированный и т.д.), высокую степень защиты, удобство работы, а также выполнять многие другие необходимые функции и услуги. Кроме этих требований функциональной полноты, к ОС предъявляется ряд важных эксплуатационных требований.

1. *Эффективность*. Под эффективностью вообще любой технической (да и не только технической) системы понимается степень соответствия системы своему назначению, которая оценивается некоторым множеством показателей эффективности [[10](https://intuit.ru/studies/courses/631/487/literature#literature.10)].

Поскольку ОС представляет собой сложную программную систему, она использует для собственных нужд значительную часть ресурсов компьютера. Часто эффективность ОС оценивают ее производительностью (пропускной способностью) – количеством задач пользователей, выполняемых за некоторый промежуток времени, временем реакции на запрос пользователя и др.

На все эти показатели эффективности ОС влияет много различных факторов, среди которых основными являются архитектура ОС, многообразие ее функций, качество программного кода, аппаратная платформа (компьютер) и др.

1. *Надежность и отказоустойчивость*. Операционная система должна быть, по меньшей мере, так же надежна, как компьютер, на котором она работает. Система должна быть защищена как от внутренних, так и от внешних сбоев и отказов. В случае ошибки в программе или аппаратуре система должна обнаружить ошибку и попытаться исправить положение или, по крайней мере, постараться свести к минимуму ущерб, нанесенный этой ошибкой пользователям.

Надежность и отказоустойчивость ОС, прежде всего, определяются архитектурными решениями, положенными в ее основу, а также отлаженностью программного кода (основные отказы и сбои ОС в основном обусловлены программными ошибками в ее модулях). Кроме того, важно, чтобы компьютер имел резервные дисковые массивы, источники бесперебойного питания и др., а также программную поддержку этих средств.

1. *Безопасность (защищенность)*. Ни один пользователь не хочет, чтобы другие пользователи ему мешали. ОС должна защищать пользователей и от воздействия чужих ошибок, и от попыток злонамеренного вмешательства (несанкционированного доступа). С этой целью в ОС как минимум должны быть средства аутентификации – определения легальности пользователей, авторизации – предоставления легальным пользователям установленных им прав доступа к ресурсам, и аудита – фиксации всех потенциально опасных для системы событий.

Свойства безопасности особенно важны для сетевых ОС. В таких ОС к задаче контроля доступа добавляется задача защиты данных, передаваемых по сети.

1. *Предсказуемость*. Требования, которые пользователь может предъявить к системе, в большинстве случаев непредсказуемы. В то же время пользователь предпочитает, чтобы обслуживание не очень сильно менялось в течение предположительного времени. В частности, запуская свою программу в системе, пользователь должен иметь основанное на опыте работы с этой программной приблизительное представление, когда ему ожидать выдачи результатов.
2. Расширяемость. В отличие от аппаратных средств компьютера полезная жизнь операционных систем измеряется десятками лет. Примером может служить ОС UNIX, да и MS-DOS. Операционные системы изменяются со временем, как правило, за счет приобретения новых свойств, например, поддержки новых типов внешних устройств или новых сетевых технологий. Если программный код модулей ОС написан таким образом, что дополнения и изменения могут вноситься без нарушения целостности системы, то такую ОС называют расширяемой. Операционная система может быть расширяемой, если при ее создании руководствовались принципами модульности, функциональной избыточности, функциональной избирательности и параметрической универсальности.
3. *Переносимость*. В идеальном случае код ОС должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которые различаются не только типом процессора, но и способом организации всей аппаратуры компьютера) одного типа на аппаратную платформу другого типа. Переносимые ОС имеют несколько вариантов реализации для разных платформ, такое свойство ОС называется также *многоплатформенностью*. Достигается это свойство за счет того, что основная часть ОС пишется на языке высокого уровня (например С, C++ и др.) и может быть легко перенесена на другой компьютер (машинно-независимая часть), а некоторая меньшая часть ОС (программы ядра) является машинно-зависимой и разрабатывается на машинном языке другого компьютера.
4. *Совместимость*. Существует несколько "долгоживущих" популярных ОС (разновидности UNIX, MS-DOS, Windows3.x, Windows NT, OS/2), для которых наработана широкая номенклатура приложений. Для пользователя, переходящего с одной ОС на другую, очень привлекательна возможность – выполнить свои приложения в новой операционной системе. Если ОС имеет средства для выполнения прикладных программ, написанных для других операционных систем, то она совместима с этими системами. Следует различать совместимость на уровне двоичных кодов и совместимость на уровне исходных текстов. Кроме того, понятие совместимости включает также поддержку пользовательских интерфейсов других ОС.
5. *Удобство*. Средства ОС должны быть простыми и гибкими, а логика ее работы ясна пользователю. Современные ОС ориентированы на обеспечение пользователю максимально возможного удобства при работе с ними. Необходимым условием этого стало наличие у ОС графического пользовательского интерфейса и всевозможных мастеров – программ, автоматизирующих активизацию функций ОС, подключение периферийных устройств, установку, настройку и эксплуатацию самой ОС.
6. *Масштабируемость*. Если ОС позволяет управлять компьютером с различным числом процессоров, обеспечивая линейное (или почти такое) возрастание производительности при увеличении числа процессоров, то такая ОС является масштабируемой. В масштабируемой ОС реализуется симметричная многопроцессорная обработка. С масштабируемостью связано понятие кластеризации – объединения в систему двух (и более) многопроцессорных компьютеров. Правда, кластеризация направлена не столько на масштабируемость, сколько на обеспечение высокой готовности системы.

Следует заметить, что в зависимости от области применения конкретной операционной системы может изменяться и состав предъявляемых к ней требований.

Производители могут предлагать свои ОС в различных, различающихся ценой и производительностью, конфигурациях. Например, Microsoft продает [[10](https://intuit.ru/studies/courses/631/487/literature#literature.10)]:

* Windows 2003 Server (до 4-х процессоров) – для малого и среднего бизнеса;
* Windows 2003 Advanced Server (до 8 процессоров, 2-узловой кластер) – для средних и крупных предприятий;
* Windows 2003 DataCenter Server (16-32 процессора, 4-узловой кластер) – для особо крупных предприятий.

## Однозадачные и многозадачные

Операционные системы различаются особенностями реализации алгоритмов управления ресурсами компьютера, областями использования и по многим другим признакам. Так, в зависимости от особенностей алгоритма управления процессором операционные системы делятся на однозадачные и многозадачные, однопользовательские и многопользовательские, на однопроцессорные и многопроцессорные, с командным или графическим интерфейсом, а также на локальные и сетевые.

Однозадачные и многозадачные операционные системы. По числу одновременно выполняемых задач операционные системы делятся на два класса:

􀂗 **однозадачные** (например, MS DOS, MSX)

􀂗 **многозадачные** (OC EC, OS\2, Unix, ОС семейства Windows) и др.

**Однозадачные ОС** в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным интерфейс пользователя с компьютером. Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

**Многозадачные ОС**, кроме вышеперечисленных функций, управляет разделением совместно используемых ресурсов, таких, как процессор, оперативная память, файлы и важнейшие устройства.

В зависимости от областей использования многозадачные ОС подразделяются на три типа:

􀂗 системы пакетной обработки (OC EC)

􀂗 системы с разделением времени (Unix, VMS, Windows, Linux);

􀂗 системы реального времени (QNX, RT\11).

Системы пакетной обработки – без непосредственного доступа пользователя, а с предварительным сбором и формированием всего блока (пакета) программ, подлежащих одновременному решению.

В системах с разделением времени каждой задаче выделяется небольшой квант процессорного времени, ни одна задача не занимает процессор надолго и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то это предполагает параллельное выполнение нескольких программ, существующих в рамках одной вычислительной системы. Ясно, что подобные системы обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая «выгодна» системе. Критерием эффективности систем с разделением времени является не максимальная пропускная способность процессора, а эффективность работы пользователя в интерактивном режиме.

Многопользовательский и однопользовательский режимы. По числу

одновременно работающих пользователей ОС могут быть разделены на однопользовательские (MS DOS, Windows 3.x) и многопользовательские (Unix, Windows NT, Windows XP, Linux). Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей.

Многопроцессорные и однопроцессорные системы. Другим важным свойством ОС является отсутствие или наличие в ней средств поддержки многопроцессорной обработки. В наши дни становится общепринятым введение в ОС функций поддержки многопроцессорной обработки.

Одним из важных признаков классификации ОС является деление их на сетевыеи локальные. Сетевые ОСпредназначены для управления ресурсами компьютера, объединенных в сеть с целью совместимого использования данных. Они представляют мощное средство разграничения доступа к информации, ее целостности и сохранности, а также другие возможности использования сетевых ресурсов. Сетевая ОС составляет основу любой вычислительной сети.

Интерфейс ОС должен быть дружелюбным к пользователю и в этом аспекте можно говорить о двух уровнях общения пользователя с машиной:

􀂃 командный интерфейс – пользователь должен прилично знать файловую систему и команды операционной системы и вводить их с клавиатуры в командную строку, имеющуюся на экране дисплея; такой интерфейс обеспечивается непосредственно MS DOS; для облегчения работы с такой ОС используют различные оболочками ОС, в частности, наиболее популярной оболочкой Norton Commander;

􀂃 интерфейс в виде графических меню – пользователь должен ориентироваться в многочисленных меню и панелях инструментов и уметь выбрать в этих меню нужные команды и файлы по условным графическим значкам, обычно сопровождаемым их наименованиями; интерфейс этого типа реализуется графическими интерфейсными системами (Windows 3.1) и операционными системами с графическим интерфейсом (Windows 9x/NT/2000/XP).

К операционным системам командного типа относятся: MS DOS, R DOS, UNIX, Linux и многие другие. Наибольшее распространение среди них получила дисковая операционная система MS DOS, как в локальном варианте, так и в составе других ОС. В качестве ОС с графическим интерфейсом можно назвать ОС семейства Windows.

Системы реального времени (ОС РВ) применяются для управления различными техническими объектами (таким, как станок, спутник, научная экспериментальная установка) или технологическими процессами (гальваническая линия, доменный процесс и т.п.). Применяют ОС РВ и в банковском деле. Критерием эффективности для систем реального времени является их способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия).

# Классификация ОС. Микроядерные и макроядерные ОС

## Макроядерныеоперационные системы

По основному архитектурному принципу операционные системы разделяются на микроядерные и макроядерные (монолитные). В некоторой степени это разделение тоже условно.

В макроядерных, или монолитных, операционных системах ядро, состоящее из множества управляющих модулей и структур данных, не разделено на центральную часть и периферийные (по отношению к этой части) модули. Ядро получается монолитным, неделимым. В этом смысле макроядерные ОС являются прямой противоположностью микроядерным. В монолитной ОС, несмотря на её возможную сильную структуризацию, очень трудно удалить один из уровней многоуровневой модульной структуры. Добавление новых функций и изменение существующих для монолитных операционных систем требует очень хорошего знания всей архитектуры ОС и чрезвычайно больших усилий.

Очень плодотворным оказался подход, основанный на модели клиент-сервер. Эта модель предполагает наличие программного компонента — потребителя какого-либо сервиса, или клиента, и программного компонента — поставщика этого сер­виса, или сервера. Взаимодействие между клиентом и сервером стандартизируется, так что сервер может обслуживать клиентов, реализованных различными способами и, возможно, разными разработчиками. При этом главным требованием является то, чтобы использовался единообразный интерфейс. Инициатором обмена обычно является клиент, который посылает запрос на обслуживание серверу, находящемуся в состоянии ожидания запроса. Один и тот же программный компонент может быть клиентом по отношению к одному виду услуг и сервером для другого вида услуг.

При поддержке монолитных операционных систем возникает ряд проблем, связанных с тем, что все компоненты макроядра работают в едином адресном пространстве. Во-первых, это опасность возникновения конфликта между различными частями ядра, во-вторых, сложность подключения к ядру новых драйверов.

Микроядерная архитектура является альтернативой классическому (монолитному) способу построения операционной системы.

Под классической архитектурой в данном случае понимается рассмотренная ранее структурная организация ОС, в соответствии с которой все основные функции операционной системы, составляющие многослойное ядро, выполняются в привилегированном режиме. При этом некоторые вспомогательные функции ОС оформляются в виде приложений и выполняются в пользовательском режиме наряду с обычными пользовательскими программами (становясь системными утилитами или обрабатывающими программами). Каждое приложение пользовательского режима работает в собственном адресном пространстве и защищено тем самым от какого-либо вмешательства других приложений. Код ядра, выполняемый в привилегированном режиме, имеет доступ к областям памяти всех приложений, но сам полностью от них защищен. Приложения обращаются к ядру с запросами (системными вызовами) на выполнение системных функций.

Суть микроядерной архитектуры состоит в следующем.

В привилегированном режиме остается работать только очень небольшая часть ОС, называемая микроядром (рис. 1). Микроядро защищено от остальных частей ОС и приложений. В состав микроядра обычно входят машинно-зависимые модули, а также модули, выполняющие базовые (но не все!) функции ядра по управлению процессами, обработке прерываний, управлению виртуальной памятью, пересылке сообщений и управлению устройствами ввода-вывода, связанные с загрузкой или чтением регистров устройств. Набор функций микроядра обычно соответствует функциям слоя базовых механизмов обычного ядра. Такие функции операционной системы трудно, если не невозможно, выполнить в пространстве пользователя.

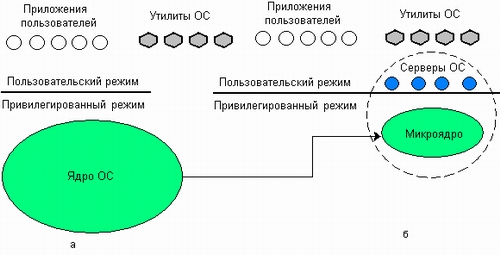


Рис. 1.Перенос основного объема функций ядра в пользовательское пространство

Все остальные более высокоуровневые функции ядра оформляются в виде приложений, работающих в пользовательском режиме. Однозначного решения о том, какие из системных функций нужно оставить в привилегированном режиме, а какие перенести в пользовательский, не существует. В общем случае многие менеджеры ресурсов, являющиеся неотъемлемыми частями обычного ядра — файловая система, подсистемы управления виртуальной памятью и процессами, менеджер безопасности и т. п., — становятся «периферийными» модулями, работающими в пользовательском режиме.

Утилиты и обрабатывающие программы вызываются в основном пользователями. Ситуации, когда одному приложению требуется выполнение функции (процедуры) другого приложения, возникают крайне редко. Поэтому в операционных системах с классической архитектурой отсутствует механизм, с помощью которого одно приложение могло бы вызвать функции другого.

Совсем другая ситуация возникает, когда в форме приложения оформляется часть операционной системы. По определению, основным назначением такого приложения является обслуживание запросов других приложений, например, создание процесса, выделение памяти, проверка прав доступа к ресурсу и т. д. Именно поэтому менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами ОС, то есть модулями, основным назначением которых является обслуживание запросов локальных приложений и других модулей ОС.

Схематично механизм обращения к функциям ОС, оформленным в виде серверов, выглядит следующим образом (рис. 2). Клиент, которым может быть либо прикладная программа, либо другой компонент ОС, запрашивает выполнение некоторой функции у соответствующего сервера, посылая ему сообщение. Непосредственная передача сообщений между приложениями невозможна, так как их адресные пространства изолированы друг от друга. Микроядро, выполняющееся в привилегированном режиме, имеет доступ к адресным пространствам каждого из этих приложений и поэтому может работать в качестве посредника. Микроядро сначала передает сообщение, содержащее имя и параметры вызываемой процедуры нужному серверу, затем сервер выполняет запрошенную операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения. Таким образом, работа микроядерной операционной системы соответствует модели клиент-сервер, в которой роль транспортных средств выполняет микроядро.

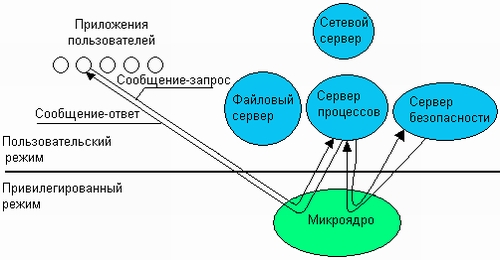


Рис. 2. Реализация системного вызова в микроядерной архитектуре

Операционные системы, основанные на концепции микроядра, в высокой степени удовлетворяют большинству требований, предъявляемых к современным ОС, обладая переносимостью, расширяемостью, надежностью и создавая хорошие предпосылки для поддержки распределенных приложений. За эти достоинства приходится платить снижением производительности, и это является основным недостатком микроядерной архитектуры.

Преимущество микроядерной архитектуры перед макроядерной заключается в том, что каждый компонент системы представляет собой самостоятельный процесс, запуск или остановка которого не отражается на работоспособности остальных процессов.

Микроядро — это минимальная главная (стержневая) часть операционной системы, служащая осно­вой модульных и переносимых расширений. Микроядро само является модулем системного программного обеспечения, работающим в наиболее приоритетном состоянии компьютера и поддерживающим связи с остальной частью операционной системы, которая рассматривается как набор серверных приложений (служб)

1. Классификация ОС. Специализированные операционные **системы**

К этому классу относятся системы реального времени; системы мобильных вычислительных устройств; специализированные сетевые системы; а также системы, предназначенные для обучения студентов. В момент возникновения персональных компьютеров их операционные системы (ДОС) также относились к этому классу.

**ДОС (Дисковые Операционные Системы)**. Как правило, это просто некий резидентный набор подпрограмм, не более того. Он загружает пользовательскую программу в память и передает ей управление, после чего программа делает с системой все, что ей заблагорассудится. Считается желательным, чтобы после завершения программы машина оставалась в таком состоянии, чтобы ДОС могла продолжить работу (принципиально же, ДОС ничем не может помешать программе привести систему в нерабочее состояние). Дисковая операционная система MS DOS для IBM PC является примером систем подобного класса. Она, правда, умеет загружать несколько программ, но не предоставляет средств для одновременного исполнения этих программ. Существование систем такого класса обусловлено их простотой и тем, что они потребляют мало ресурсов. Еще одна причина, по которой такие системы могут использоваться даже на довольно мощных машинах – требование программной совместимости с ранними моделями того же семейства компьютеров.

**Системы реального времени** – системы, предназначенные для облегчения разработки так называемых приложений реального времени. Это программы, управляющие некомпьютерным по своей природе оборудованием, часто с очень жесткими ограничениями по времени реакции. Примером такого приложения может быть программа бортового компьютера крылатой ракеты, системы управления ускорителем элементарных частиц или промышленным оборудованием. Такие системы обязаны поддерживать многопроцессность, гарантированноевремя реакции на внешнее событие, простой доступ к таймеру и внешним устройствам. Примером такой системы может служить ОС QNX. Любопытно, что multimedia при качественной реализации предъявляет к системе те же требования, что и промышленные задачи реального времени. В multimedia основной проблемой является синхронизация изображения на экране со звуком. Именно в таком порядке. Звук обычно генерируется внешним аппаратным устройством с собственным таймером, и изображение синхронизируется с ним же. Человек способен заметить довольно малые временные неоднородности в звуковом потоке. Напротив, пропуск кадров в визуальном потоке не так заметен, а расхождение звука и изображения заметно уже при задержках около 30 мс. Поэтому системы качественного multimedia должны обеспечивать синхронизацию с такой же или более высокой точностью, что мало отличается от систем мягкого реального времени.

**Системы для обучения студентов**. Во времена молодости UNIX (версия 6) ее исходные коды были широко доступны по лицензии AT&T и активно изучались. Джон Лайонс (John Lions) из университета Нового Южного Уэльса в Австралии даже написал небольшую брошюру, шаг за шагом описывающую работу UNIX. С разрешения AT&T эта брошюра использовалась во многих университетских курсах по операционным системам. С выходом версии 7 система UNIX превратилась в дорогостоящий коммерческий продукт. Лицензия, под которой она распространялась, запрещала преподавание исходного кода на учебных курсах, чтобы не подвергать риску его статус коммерческого секрета. Поэтому многие университеты просто прекратили изучение UNIX, довольствуясь одной теорией. Чтобы исправить ситуацию Эндрив Таненбаум в 1987 году написал собственную операционную систему MINIX (mini-UNIX), предназначенную для обучения студентов, которая с точки зрения пользователя совместима с UNIX, но внутри совершенно самостоятельна. Ядро этой системы имело всего 4000 строк кода, в то время как в UNIX, или в Windows – это миллионы строк кода. Система настолько мала, что даже начинающий мог понять, как она работает. Одним из пользователей MINIX был финский студент по имени Линус Торвальдсен. Он установил ее на свой компьютер и тщательно изучил исходный код. Опыты по усовершенствованию этой системы привели к созданию операционной системы LINUX в 1991 году.

Сетевые системы. Этот термин употребляют в двух различных смыслах:

1. Системы, предназначенные только для предоставления сетевых услуг, аналогично тому, как ДОС предназначена для предоставления средств работы с диском. Под такое понимание подходят узкоспециализированные системы, такие как Novell Netware или, например, программное обеспечение маршрутизаторов Cisco.

2. Системы, способные предоставят сетевые услуги. Под такое определение подходят практически все современные ОС общего назначения.

# Классификация ОС. Операционные системы общего назначения

## Операционные системы общего назначения

Различают три типа операционных систем (ОС) общего назначения: поддерживающие однопрограммный режим работы и диалоговый способ общения, обеспечивающие пакетную обработку задач в режиме мультипрограммирования и операционные системы разделения времени.  
  
1. Операционные системы общего назначения, поддерживающие однопрограммный режим работы и диалоговый способ общения включают в себя средства, обеспечивающие ввод и вывод информации, управляют работой системных обрабатывающих программ - трансляторов, редакторов, предоставляют пользователю сведения о ходе выполнения задач, обеспечивают работу с библиотеками. Обычно такие операционные системы называют мониторными. Они не повышают производительности ЭВМ, но позволяют программисту вмешиваться в ход выполнения задания, что резко повышает производительность его работы, особенно на этапе отладки программ.  
  
2 Операционные системы общего назначения, обеспечивающие пакетную обработку задач в режиме мультипрограммирования применяются в ВС средней и большой производительности. В RAM ЭВМ одновременно находится несколько системных и пользовательских задач, и когда одна из них обрабатывается процессором, то для остальных осуществляются необходимые обмены с внешним устройством (ВУ).  
  
Эффективность использования ВС при этом во многом зависит от состава пакета задач, подлежащих выполнению, так как могут возникать ситуации, когда все задачи находятся в состоянии ожидания и процессор простаивает ( в условиях потока отладочных задач, каждая из которых характеризуется многократными обменами и незначительным временем, затрачиваемым собственно на счет). Эффективность работы пользователя при этом невысокая, так как в условиях пакетной обработки задач он не имеет возможности вмешиваться в процесс выполнения своей программы.  
  
Рассмотрим основные функции ОС общего назначения, обеспечивающей мультипрограммный режим обработки задач.  
  
Операционная система должна выполнять рациональное планирование работ по обработке всех поступающих задач (комплекс мероприятий по вводу задач в ЭВМ, распознаванию их характеристик, размещению всех входных наборов данных на внешних носителях, организации входных и выходных очередей).  
  
Как правило, задачи из входного потока данных, прочитанного одним из внешних устройств (ВУ), не сразу попадают в RAM ЭВМ, а размещаются на устройствах внешней памяти. В режимах пакетной обработки задачи выстраиваются в очередь (входную очередь), место задачи в очереди определяется ее приоритетом. Перенос задачи из очереди в RAM ЭВМ происходит автоматически.  
  
При реализации комплекса мероприятий, выполняемого ОС непосредственно перед началом решения задачи, главное внимание уделяется предоставлению всех необходимых для решения задачи ресурсов ВС (области RAM, места на диске, требующихся наборов данных и т.п.)  
Если для решения очередной задачи не хватает ресурсов, ОС должна принять одно из следующих решений:

* отобрать часть ресурсов у какой - либо другой задачи, выполнявшейся в данный момент и менее приоритетной;
* подождать, пока какая-нибудь из решаемых задач завершится и освободит требуемый ресурс;
* пропустить вне очереди ту задачу, чья очередь еще не подошла, но для выполнения которой ресурсов достаточно.

3. Операционные системы разделения времени позволяют реализовать возможность повышения производительности труда пользователя за счет его доступа к своей задаче в процессе ее выполнения и повышения производительности ВС за счет мультипрограммирования. Режим разделения времени создает иллюзию одновременного доступа нескольких пользователей ко всем вычислительным ресурсам ВС. Каждый пользователь общается с системой так, как если бы ему одному принадлежали все вычислительные ресурсы: он может остановить выполнение своей задачи в нужном месте, просмотреть требуемые области RAM, с заданного места выполнить свою программу по командам и т.д. На самом же деле каждый пользователь получает для своей задачи достаточную зону RAM, процессор и прочие вычислительные ресурсы только в течение определенного и достаточно малого интервала времени, как уже говорилось выше - кванта.  
  
Пропускная способность ВС в режиме разделения времени ниже, чем при обработке задач в режиме мультипрограммирования, из-за накладных расходов ОС, вызванных частыми переключениями процессора и главным образом многочисленными переносами задач из RAM на жесткий диск и обратно, то есть свопингами. Во многих пользовательских системах режим разделения времени сочетается с пакетной обработкой задач в режиме мультипрограммирования. В этом случае RAM ЭВМ разделяется на зону для пакетной обработки и на зону (или несколько зон в зависимости от емкости RAM) для выполнения задач в режиме разделения времени. Такое сочетание позволяет загружать процессор даже в ситуациях, когда все пользователи режима разделения времени остановят выполнение своих задач. Такие системы используются при решении научно- технических задач. При этом главное назначение таких ОС - обеспечение более высокой эффективности использования всех вычислительных ресурсов ВС и достижение максимальных удобств в работе пользователя. Однако использование операционных систем общего назначения в условиях работы конкретного пользователя часто означает явную избыточность многих системных средств. В таких случаях применяют ОС специального назначения.

# Классификация ОС. Системы с асимметричной процессорной обработкой

Асимметричная мультипроцессорная система (asymmetric multiprocessing) – это

многопроцессорная компьютерная система, в которой процессоры специализированы по своим функциям.

Каждому процессору дается специфическое задание; главный процессор (master processor) планирует

работу подчиненных процессоров (slave processors).В такой системе ОС, как правило, выполняется

на одном определенном, закрепленном за ней, центральном процессоре. Подобная архитектура более

типична для очень больших систем. Пример – система "Эльбрус", которая имела в своем составе, в

зависимости от конфигурации, от одного до 10 центральных процессоров, от одного до четырех

специализированных процессоров ввода вывода (ПВВ),от одного до четырех процессоров передачи

данных (ППД).

# Классификация ОС. Системы с симметричной процессорной обработкой

Симметричная мультипроцессорная система - symmetric multiprocessing (SMP) – это

многопроцессорная компьютерная система, все процессоры которой равноправны и используют одну и ту

же копию ОС. Операционная система при этом может выполняться на любом процессоре. В такой системе

любому свободному процессору может быть поручено любое задание. Все процессоры используют общую

память и общие дисковые ресурсы. Несколько процессов (или потоков) могут исполняться одновременно

без существенного нарушения производительности. Большинство современных ОС поддерживают

архитектуру SMP. После инсталляции ОС (например, Linux) на симметричную мультипроцессорную систему

пользователь может заметить в меню boot loader, что фактически на его компьютер установилась не одна,

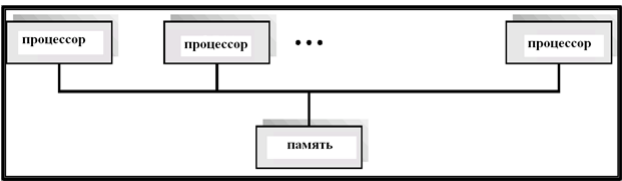
а две версии ОС – с поддержкой SMP и без нее.

Рис. 3.1. Схема организации SMP-архитектуры компьютеров

# Процессы. Граф существования процесса (ТОЛЬКО ТУТ ПРО САМИ ПРОЦЕССЫ)

///////////////////

**Процесс (process)** - это пользовательская программа при ее исполнении в компьютерной системе. Для выполнения процесса требуется ряд ресурсов, включая время процессора, память, файлы, устройства ввода-вывода, сетевые устройства и др.

//////////////////

**Определение 1**. Процесс — это выполняющаяся программа.

**Определение 2**. Процесс — это некоторая деятельность,

связанная с исполнением программы на процессоре.

**Определение 3** (ГОСТ19781-83). Процесс — это система

действий, реализующая определенную функцию в

вычислительной системе и оформленная так, что

управляющая программа вычислительной системы может

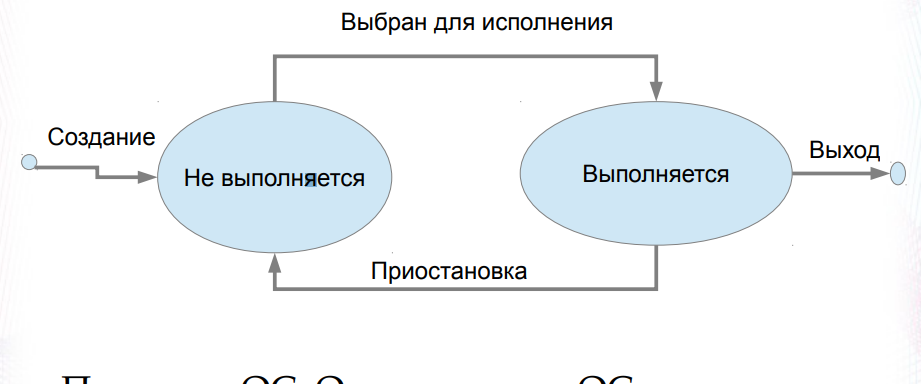
перераспределять ресурсы этой системы в целях обеспечения

многозадачности.

Требования к ОС:

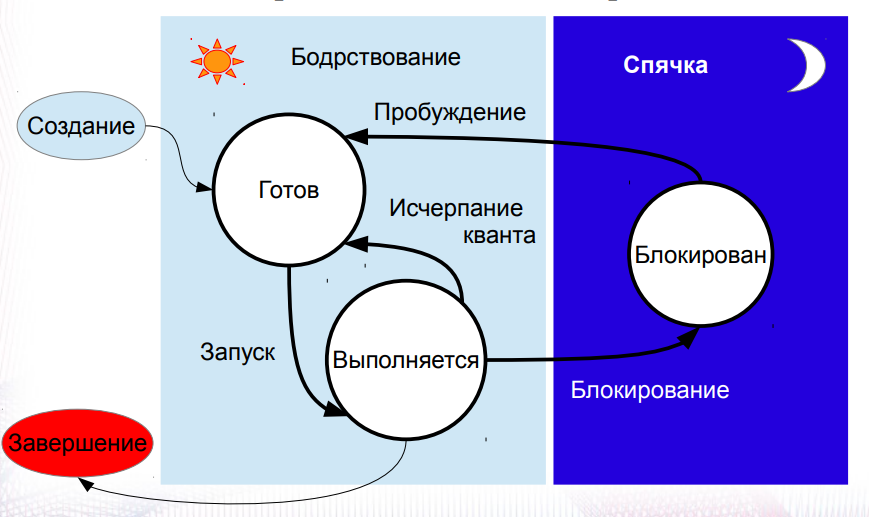
* Чередование выполняющихся процессов
* Распределение ресурсов между процессами
* Обеспечение взаимодействия между процессами

**Модель процесса с двумя состояниями**



Пакетные ОС, Однозадачные ОС

**Диаграмма состояния процесса**



**Блок Управления Процессом**

**1. Идентификаторы**

**(a) Идентификатор процесса**

**(b) Идентификатор родительского процесса**

**(c) Идентификатор пользователя/группы**

**2. Контекст (состояние процессора)**

**(a) Программно-доступные регистры**

**(b) Регистры состояния и адресные регистры процессора**

**3. Управляющая информация процесса**

**(a) Состояние процесса**

**(b) Приоритет процесса**

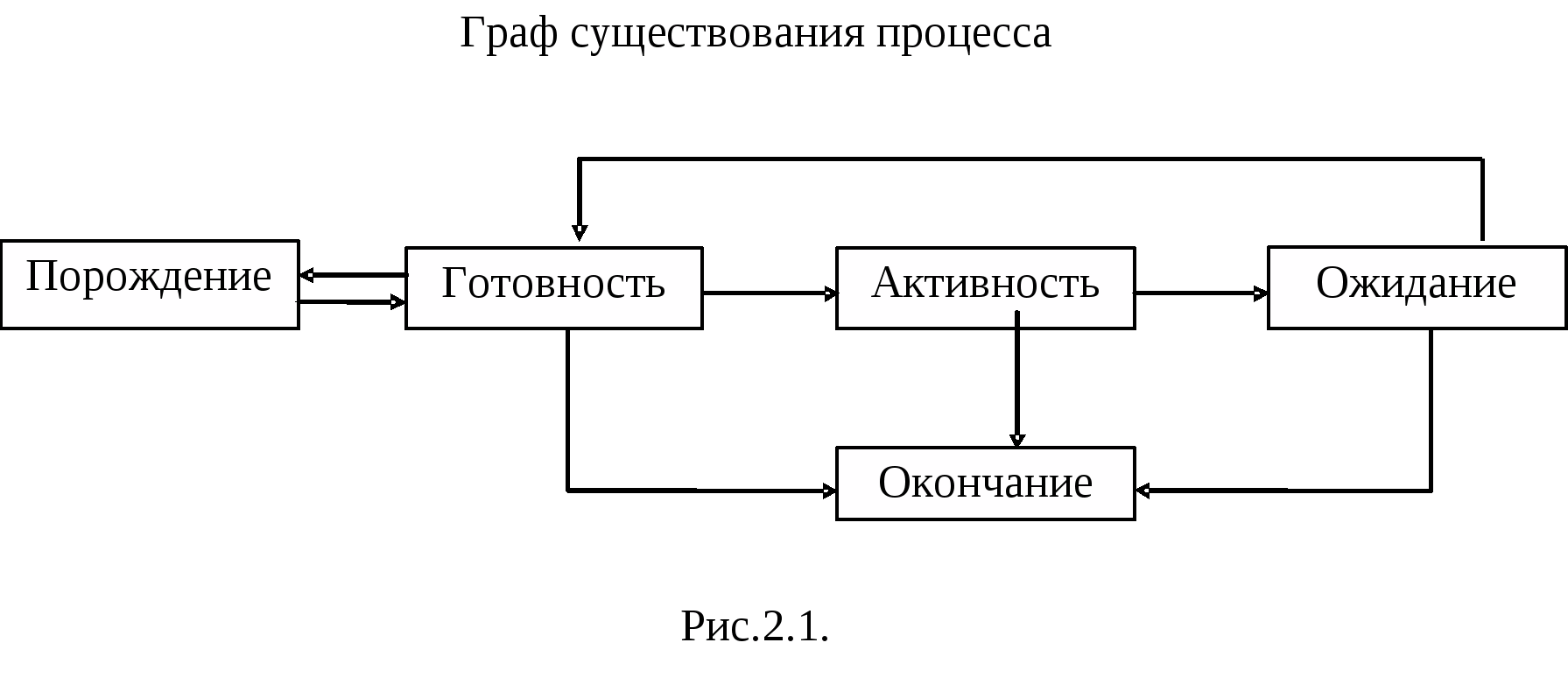
**(c) Привилегии/Права**

**(d) Ресурсы (открытые файлы, среда, распределение**

**памяти, ...)**

**Управление Процессами**

* **Создание процесса**
* **Переключение**
* **Приостановка/активация**
* **Завершение**



# Процессы. Классификация процессов

1. **В зависимости от ИНТЕРВАЛОВ СУЩЕСТВОВАНИЯ процессы бывают:**

**Процессы реального времени** – это такой вид процессов, требующие такого планирования, при котором гарантировалось бы окончание процесса до наступления некоторого конкретного момента времени.

**Интерактивные процессы**– процессы, время существования которых должно быть не более интервала времени допустимой реакции ЭВМ на запросы пользователя.

**Пакетные процессы**– все остальные процессы.

1. **ПО ГЕНЕАЛОГИЧЕСКОМУ признаку:**

**Порождающие** – вид процессов, задающий требования для порождения других процессов.

**Порожденные**– процессы, создаваемые по требованию других процессов.

1. **ПО РЕЗУЛЬТАТИВНОСТИ:**

**Эквивалентные**– процессы, имеющие одинаковых конечный результат обработки одних и тех же исходных данных по одной и той же или даже различным программам на одном и том же или различных процессах.

**Тождественные**– эквивалентные процессы, в которых обработка данных происходит по одной и той же программе, но трассы которых не совпадают.

**Равные**– тождественные процессы, трассы которых совпадают.

**Различные**– все остальные процессы.

1. **По ДИНАМИЧЕСКОМУ признаку:**

**Последовательные**– процессы, интервалы которых не пересекаются во времени.

**Параллельные***–*процессы, существующие одновременно на рассматриваемом промежутке времени.

**Комбинированные**– процессы, существующие на рассматриваемом интервале хотя бы в одной точке, но не существующие в других.

1. **По ПРИНАДЛЕЖНОСТИ К ЦЕНТРАЛЬНОМУ ПРОЦЕССОРУ:**

**Внутренние (программные***)* – процессы, развивающиеся на центральном процессоре.

**Внешние**– процессы, развитие которых происходит под контролем или управлением операционных систем на процессорах, отличных от центрального.

1. **По ПРИНАДЛЕЖНОСТИ К ОПЕРАЦИОННЫМ СИСТЕМАМ:**

**Системны***е –*процессы, при развитии которых исполняется программа из состава операционной системы.

**Пользовательские**– процессы, при развитии которых исполняется пользовательская (прикладная) программа.

1. **ПО СВЯЗАННОСТИ:**

**Взаимосвязанные**– процессы, между которыми, с помощью системы управления процессами, поддерживаются связи какого-либо рода: функциональные, пространственно-временные, управляющие, информационные и т.д.

**Изолированные** – процессы со слабыми связями.

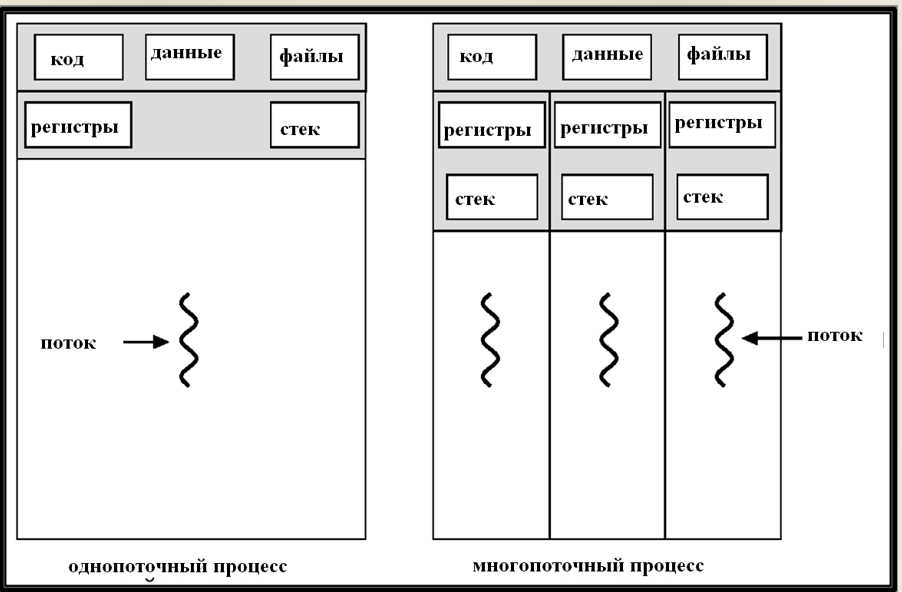
**Информационно независимые**– взаимосвязанные процессы, при развитии которые используют совместно некоторые ресурсы, но информационно не связанные между собой. Связь между такими процессами может быть либо функциональная, либо пространственно-временная.

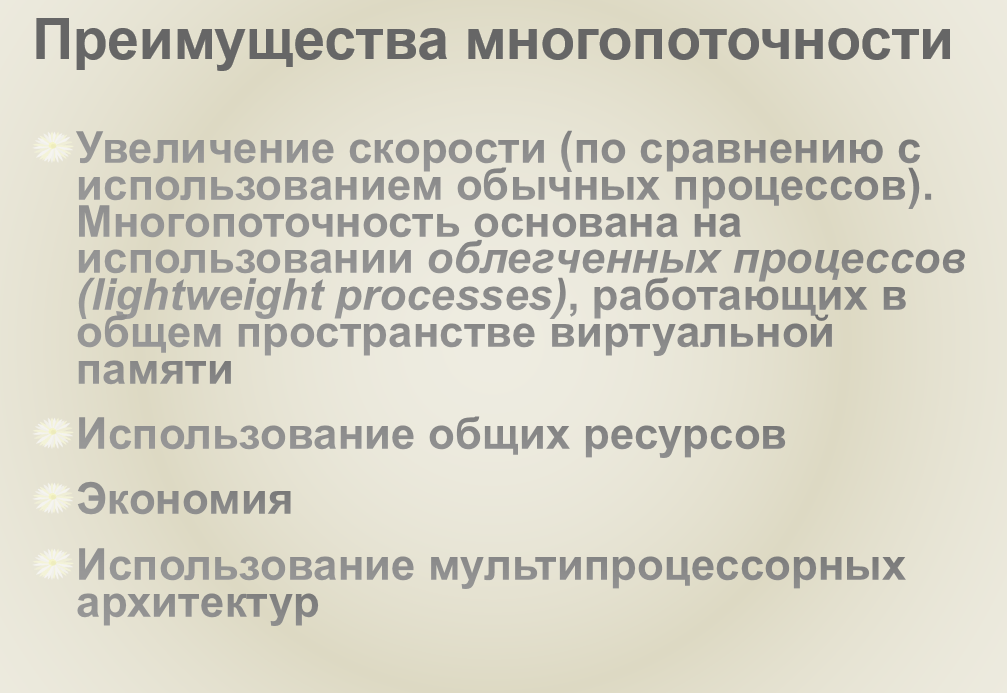
**Взаимодействующие**– процессы, имеющие информационные связи.

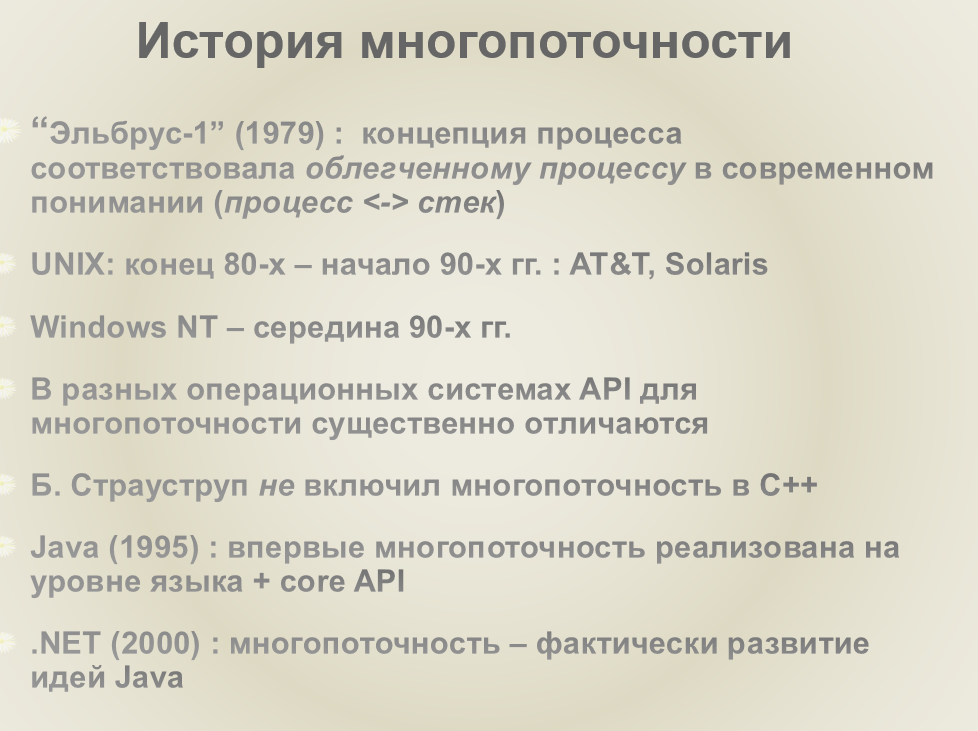
**Конкурирующие**– взаимосвязанные процессы, имеющие связи по ресурсам.

////////

# Процессы. Программные потоки





**Модели многопоточности  
(каким образом пользовательские потоки отображаются в потоки ядра?)**

**Много / Один (Many-to-One)**

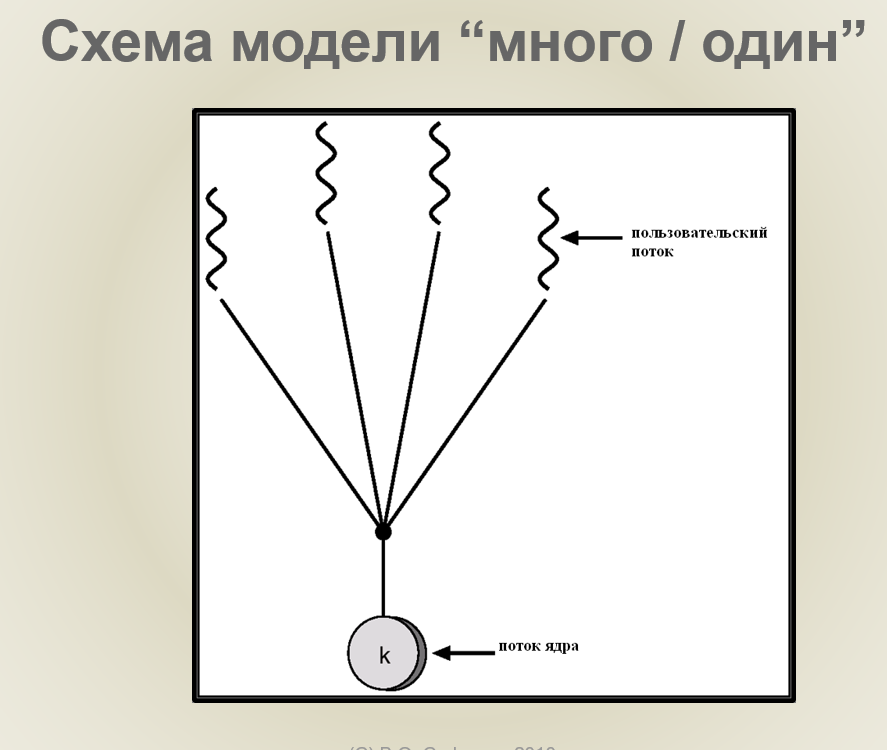
**Один / Один (One-to-One)**

**Много / Много (Many-to-Many)**

**Модель “много / один”**

**Несколько потоков пользовательского уровня отображаются в один системный поток**

**Используется в системах, которые не поддерживают множественные системные потоки**

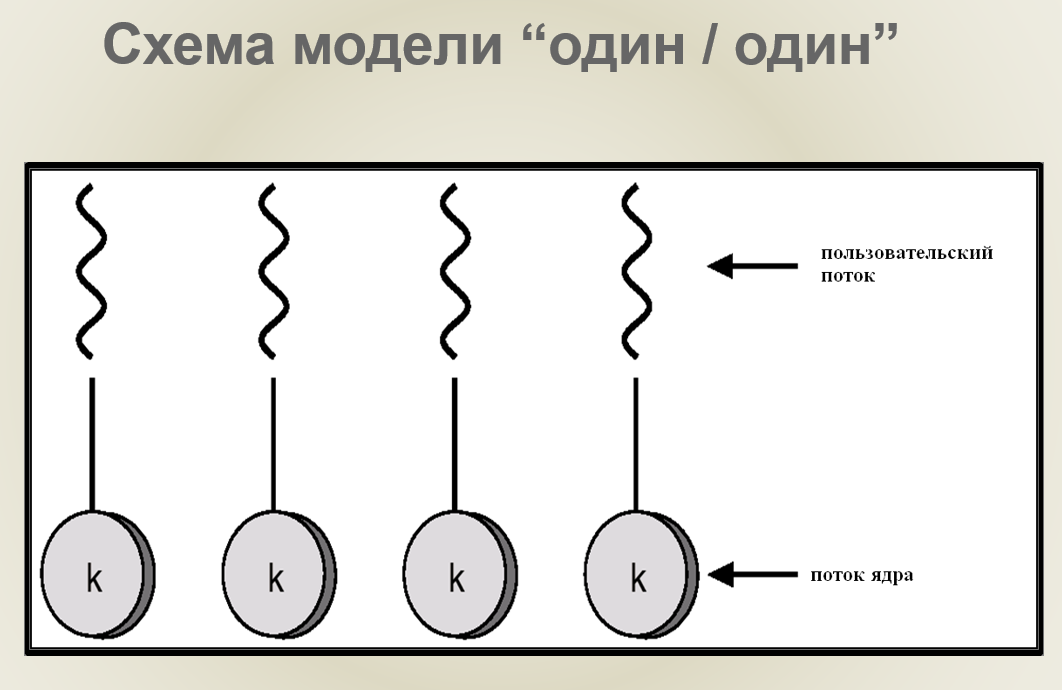


**Модель “один / один”**

* **Каждый поток пользовательского уровня отображается в один системный поток**
* **Примеры**

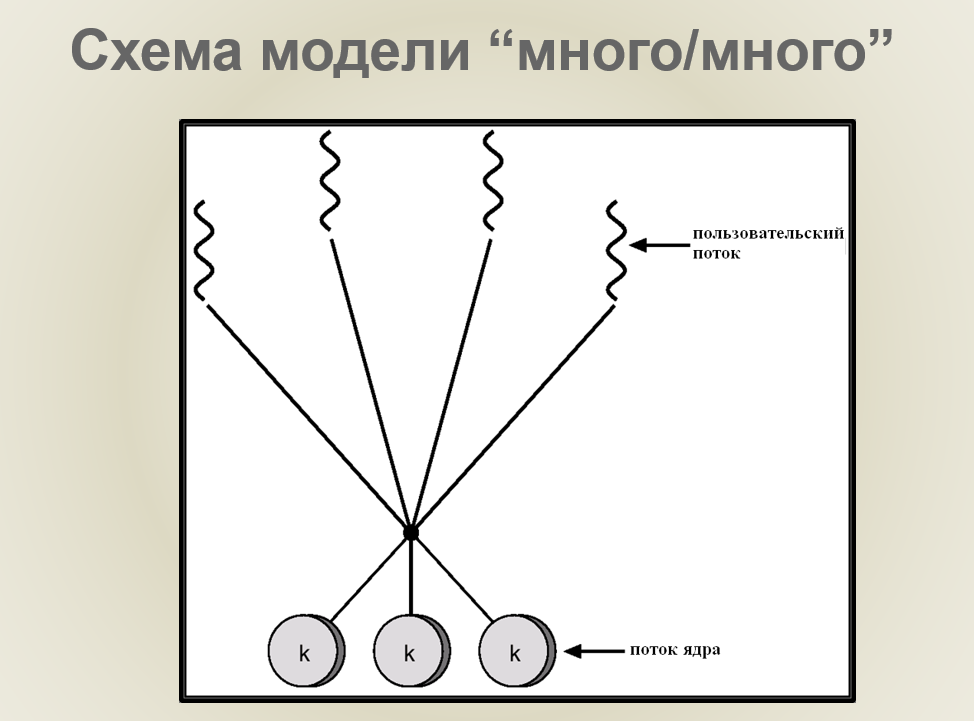
**- Windows 95/98/NT/2000**

**- OS/2**



**Модель “много / много”**

* **Допускает, чтобы несколько потоков пользовательского уровня могли отображаться в несколько системных потоков**
* **Позволяет ОС создавать достаточно большое число системных потоков.**
* **Solaris**
* **Windows NT/2000 с пакетом *ThreadFiber***



**Проблемы многопоточности:**

* **Семантика системных вызовов fork() и exec()**
* **Прекращение потоков**
* **Обработка сигналов**
* **Группы потоков**
* **Локальные данные потока (thread-local storage)**
* **Синхронизация потоков**
* **Тупики (deadlocks) и их предотвращение**

**PThreads**

* **POSIX – Portable Operating Systems Interface of uniX kind**
* **Стандарт POSIX (IEEE 1003.1c) - API для создания и синхронизации потоков**
* **API определяет поведение библиотеки потоков. Реализация – на усмотрение авторов библиотеки.**
* **Распространены в ОС типа UNIX.**

# Управление ресурсами. Состояние процесса

# Сохранение состояния процесса

**Состояния процесса**

При исполнении процесс может изменять свое состояние следующим образом:

**Новый** (new):Процесс создается операционной системой, но еще не начал выполняться.

**Исполняемый** (running):Исполняются команды процесса на процессоре или процессорах компьютерной системы под управлением ОС.

**Ожидающий** (waiting):Процесс ожидает наступления некоторого события, например, завершения вводавывода. В состоянии ожидания процесс не занимает процессор.

**Готовый к выполнению** (ready):Процесс ожидает получения ресурсов процессора для его исполнения. В состояние готовности к выполнению процесс попадает обычно либо при его создании, либо после завершения ввода-вывода (из состояния ожидания).

**Завершенный** (terminated):Исполнение процесса завершено.

# Потоки. Средства синхронизации потоков

История

* Совместный доступ к общим данным может привести к нарушению их целостности (inconsistency).
* Поддержание целостности общих данных требует механизмов упорядочения работы взаимодействующих процессов (потоков).
* Решение проблемы общего буфера с помощью глобальной (общей) памяти допускает, чтобы не более чем *n* – 1 элементов данных могли быть записаны в буфер в каждый момент времени.
  + Предположим, что в системе производитель/потребитель мы модифицируем код, добавляя переменную *counter*, инициализируемую 0 и увеличиваемую каждый раз, когда в буфер добавляется новый элемент данных

//////////////////////////////////////////////////////

**Цели и средства синхронизации**

Потребность в синхронизации потоков возникает в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходи­ма для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Во многих операционных системах эти средства называются средствами межпроцессного взаимодействия - Inter Process Communications (IPC), что отражает историческую первичность понятия «процесс» по отношению к понятию «поток». Обычно к средствам IPC относят не только средства межпроцессной синхрони­зации, но и средства межпроцессного обмена данными.

Потоки в общем случае (когда программист не предпринял спе­циальных мер по их синхронизации) протекают независимо, асинхронно друг другу. Это справедливо как по отношению к потокам одного процесса, выпол­няющим общий программный код, так и по отношению к потокам разных про­цессов, каждый из которых выполняет собственную программу.

Любое взаимодействие процессов или потоков связано с их синхронизацией, ко­торая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступ­лении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными. Например, поток-получатель должен обращаться за данными только после того, как они помещены в буфер потоком-отправителем. Если же поток-получатель обратился к данным до момента их поступления в буфер, то он должен быть приостановлен.

Для синхронизации потоков прикладных программ программист может исполь­зовать как собственные средства и приемы синхронизации, так и средства опера­ционной системы.

Обычно разработчики операционных систем предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, а также быть функционально специализированными, например средства для синхронизации потоков одного процесса, средства для синхронизации потоков разных процессов при обмене данными и т. д. Часто функциональные возможности разных системных вызовов синхронизации пере­крываются, так что для решения одной задачи программист может воспользо­ваться несколькими вызовами в зависимости от своих личных предпочтений.

**Критическая секция**

Критическая секция – это часть программы, результат выпол­нения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда вы­полнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным критическим данным, при несогласованном изменении которых могут возникнуть нежелательные эффекты.

Чтобы исключить эффект гонок по отношению к критическим данным, необхо­димо обеспечить, чтобы в каждый момент времени в критической секции свя­занной с этими данными, находился только один поток. При этом неважно, находится этот поток в активном или в приостановленном состоянии. Этот прием называют взаимным исключением. Операционная система использует разные способы реализации взаимного исключения.

**Блокирующие переменные**

Для синхронизации потоков одного процесса прикладной программист может использовать глобальные блокирующие переменные. С этими переменными, к ко­торым все потоки процесса имеют прямой доступ, программист работает, не об­ращаясь к системным вызовам ОС.

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рисунке 3.11 показан фрагмент алго­ритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную F(D). Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь поток с данными D. Если переменная F(D) установлена в 0, то данные заняты и проверка циклически повторяется. Если же данные свободны (F(D) = 1), то значение пере­менной F(D) устанавливается в 0 и поток входит в критическую секцию. После того как поток выполнит все действия с данными D, значение переменной F(D) снова устанавливается равным 1.

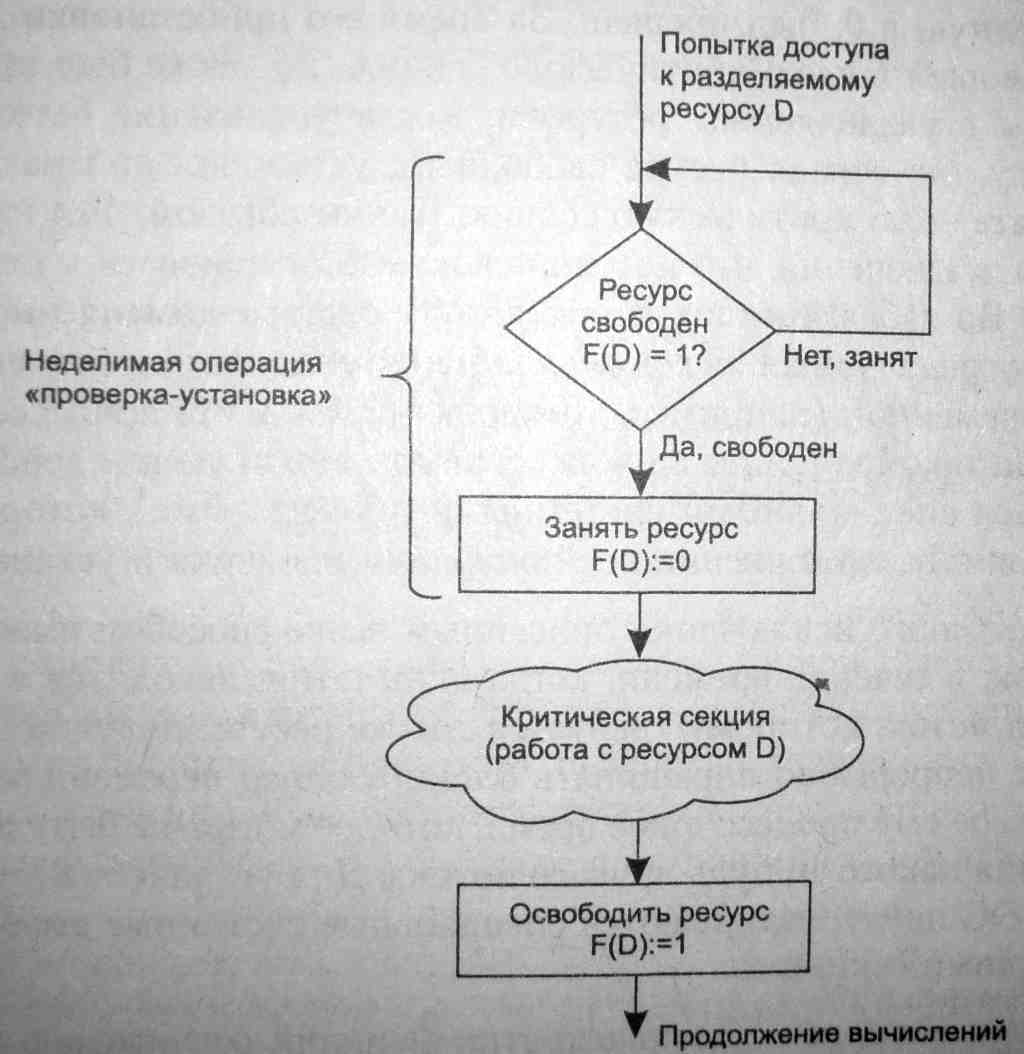


Рисунок 3.11 – Реализация критических секций с использованием блокирующих переменных

**Семафоры**

Семафоры Дийкстры – обобщение блокирующих переменных. Вместо двоичных переменных Дийкстра (Dijkstra) предложил использовать переменные, которые могут принимать целые неотрицательные значения. Такие переменные, используемые для синхронизации вычислительных процессов, получили название семафоров.

Для работы с семафорами вводятся два примитива, традиционно обозначаемых P и V. Пусть переменная S представляет собой семафор. Тогда действия V(S) и P(S) определяются следующим образом:

1. V(S): переменная S увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции;
2. P(S): уменьшение S на 1, если это возможно. Если S = 0 и невозможно уменьшить S, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию Р, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Никакие прерывания во время выполнения примитивов V и Р недопустимы.

Семафоры позволяют эффективно решать задачу синхронизации доступа к ресурсным пулам, таким, например, как набор идентичных в функцио­нальном назначении внешних устройств (модемов, принтеров, портов), или на­бор областей памяти одинаковой величины, или информационных структур. Во всех этих и подобных им случаях с помощью семафоров можно организовать доступ к разделяемым ресурсам сразу нескольких потоков.

Семафор может использоваться и в качестве блокирующей переменной. Обеспечим взаимное исключение с помощью двоичного семафора b, что показано на рисунке 3.12. Оба потока после проверки доступности буферов должны выполнить проверку доступности критической секции.

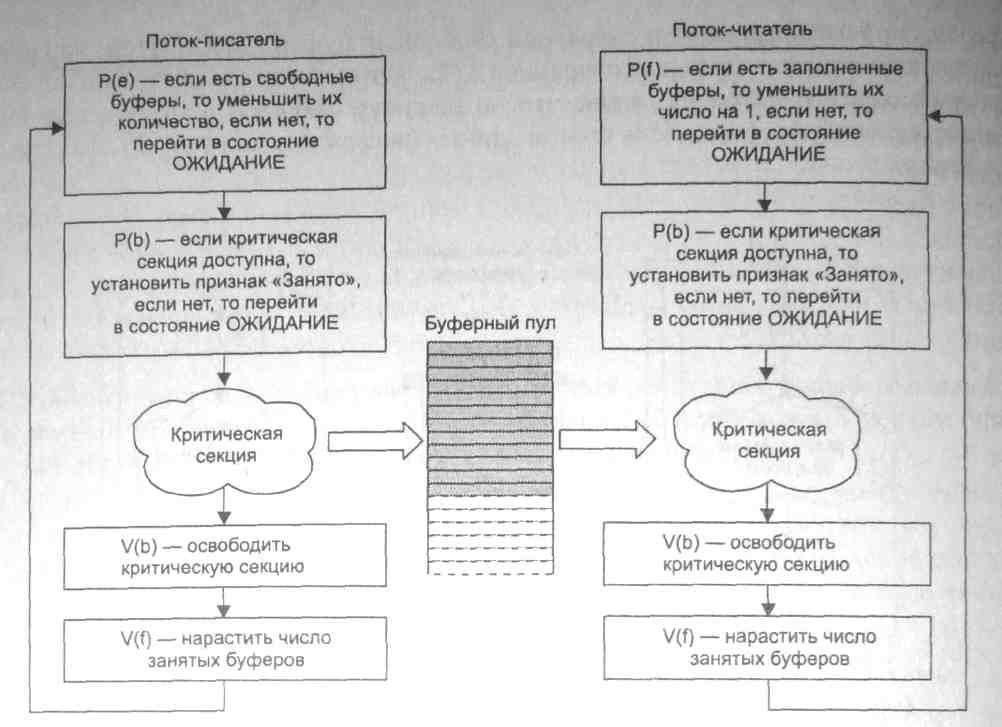


Рисунок 3.12 – Использование двоичного семафора.

Тупики

Приведенный выше пример позволяет также проиллюстрировать еще одну проблему синхронизации — взаимные блокировки, называемые также дедлоками (deadlocks), клинчами (clinch), или тупиками*.*Покажем, что если переставить местами операции Р(е) и Р(b) в потоке-писателе, то при некотором стечении обстоя­тельств эти два потока могут взаимно блокировать друг друга.

Итак, пусть поток-писатель начинает свою работу с проверки доступности критической секции — операции Р(b), и пусть он первым войдет в критическую секцию. Выполняя операцию Р(е), он может обнаружить отсутствие свободных бу­феров и перейти в состояние ожидания. Из этого состоя­ния его может вывести только поток-читатель, который возьмет очередную за­пись из буфера. Но поток-читатель не сможет этого сделать, так как для этого ему потребуется войти в критическую секцию, вход в которую заблокирован по­током-писателем. Таким образом, ни один из этих потоков не может завершить начатую работу и возникнет тупиковая ситуация, которая не может разрешиться без внешнего воздействия.

В рассмотренном примере тупик был образован двумя потоками, но взаимно блокировать друг друга может и большее число потоков. На рисунке 3.13 показано такое распределение ресурсов Ri между несколькими потоками Tj, которое при­вело к возникновению взаимных блокировок. Стрелки обозначают потребность потока в ресурсах. Например, потоку Т1 для выполнения работы необходимы ресурсы R1 и R2, из ко­торых выделен только один — R1, а ресурс R2 удерживается потоком Т2. Ни один из четырех показанных на рисунке потоков не может продолжить свою работу, так как не имеет всех необходимых для этого ресурсов.

Невозможность потоков завершить начатую работу из-за возникновения взаимных блокировок снижает производительность вычислительной системы. По­этому проблеме предотвращения тупиков уделяется большое внимание. На тот случай, когда взаимная блокировка все же возникает, система должна предоста­вить администратору-оператору средства, с помощью которых он смог бы распо­знать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. Если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

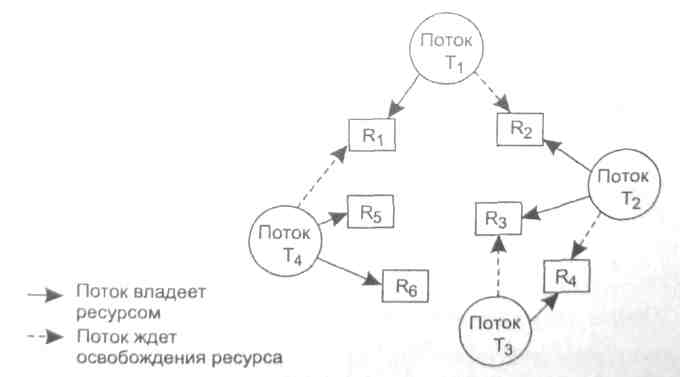


Рисунок 3.13 – Взаимная блокировка нескольких потоков.

# Диспетчеризация процессора. Основная цель диспетчеризации процессора

**Планировщик процессора (scheduler)**

**Ии тетради**

**[Диспетчеризация процессора –** распределение его времени между процессами в системе.

Цель диспетчеризации – максимальная загрузка процессора.

Планировщик – компонент ОС, которая выбирает 1 из нескольких процессов, загруженных в память и готовых к выполнению и выделяет процессов для 1 из них(или для него. не ебу)

**]**

* Выбирает один из нескольких процессов, загруженных в память и готовых к выполнению, и выделяет процессор для одного из них.
* Решения по диспетчеризации могут быть приняты, когда процесс:

1. Переключается из состояния выполнения в состояние ожидания.

2. Переключается из состояния выполнения в состояние готовности к выполнению.

3. Переключается из состояния ожидания в состояние готовности.

4. Завершается.

* Диспетчеризация типов 1 и 4 – *без прерывания процесса*

*(non-preemptive)*.

* В остальных случаях – *с прерыванием процесса (preemptive).*

**Собственно диспетчер**

* Модуль диспетчера предоставляет процессор тому процессу, который был выбран планировщиком, то есть:
  + Переключает контекст
    - Переключает процессор в пользовательский режим
    - Выполняет переход по соответствующему адресу в пользовательскую программу для ее рестарта
* Скрытая активность диспетчера (*dispatch latency)* – время, требуемое для диспетчера, чтобы остановить один процесс и стартовать другой.
* **Критерии диспетчеризации**
* Использование процессора – поддержание его в режиме занятости, насколько это возможно
* Пропускная способность (throughput) – число процессов, завершающих свое выполнение за единицу времени
* Время обработки (turnaround time) – время, необходимое для исполнения какого-либо процесса
* Время ожидания (waiting time)– время, которое процесс ждет в очереди процессов, готовых к выполнению
* Время ответа (response time) – время, требуемое от момента первого запроса до первого ответа (для среды разделения времени)
* **Критерии оптимизации**
* **Max CPU utilization**
* **Max throughput**
* **Min turnaround time**
* **Min waiting time**
* **Min response time**

# Планировщик. Разновидности стратегий, с точки зрения прерывания или избежания прерывания процессов, использует планировщик(ТУТ ПЛАНИРОВЩИК)

ИЗ ТЕТРАДИ

[

Планировщик – компонент ОС, которая выбирает 1 из нескольких процессов, загруженных в память и готовых к выполнению и выделяет процессов для 1 из них(или для него. не ебу)

]

Планировщики, выполняющие диспетчеризацию процессов

В операционной системе диспетчеризация процессов выполняется обычно несколькими

планировщиками,каждый из которых имеет свою периодичность вызовов и свою определенную задачу,

которую он решает.

Долговременный планировщик (планировщик заданий) определяет, какие процессы должны быть

перемещены в очередь готовых процессов.

Кратковременный планировщик (планировщик процессора) – определяет, какие процессы должны

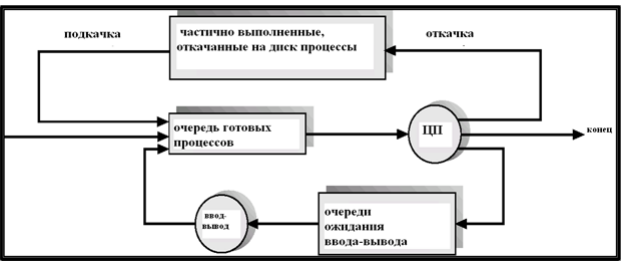
быть выполнены следующими и каким процессам должен быть предоставлен процессор.

Для реализации режима разделения времени в систему может быть добавлен также планировщик

откачки и подкачки процессов, определяющий, какие пользовательские процессы должны быть

подкачаны в память или откачаны на диск. Схема работы системы, включающей такой планировщик,

изображена на рис. 8.6

Рис. 8.6. Добавление в систему планировщика откачки и подкачки процессов.

Особенности планировщиков и процессов.Каждый планировщик имеет свои особенности поведения,

как и каждый процесс.

Кратковременный планировщик вызывается очень часто, по крайней мере не реже, чем по истечение

очередного кванта времени процессора. Поэтому он должен быть очень быстрым, максимально

эффективно реализованным. Понятно, что недопустимо, например, если время работы этого планировщика

окажется сравнимым с размером самого кванта времени – слишком велики будут накладные расходы.

Долговременный планировщик вызывается относительно редко, так как система не столь часто

принимает решения о переводе процесса в очередь готовых процессов. Поэтому он может быть

сравнительно медленным, не столь эффективно реализованным.

Однако, поскольку основной задачей системы в целом остается обслуживание как можно большего числа

процессов, именно долговременный планировщик определяет степень (коэффициент)

мультипрограммирования – число процессов, которое обслуживает система в единицу времени.

Сами процессы также могут вести себя по-разному, с точки зрения их диспетчеризации. Процессы могут

быть:

**Ориентированными на ввод-вывод** (I/O-bound) – процессы, которые тратят больше времени на

ввод-вывод, чем на вычисления. Такие процессы обычно расходуют много коротких квантов

процессорного времени.

**Ориентированные на использование процессора** (CPU-bound) – процессы, которые тратят

основное время на вычисления. Такие процессы расходуют небольшое число долговременных квантов

процессорного времени.

Переключение контекста

Когда процессор переключается на другой процесс, система должна сохранить состояние старого процесса

и загрузить сохраненное состояние для нового процесса. Такое действие системы называется

переключением контекста (context switch).

Переключение контекста относится к накладным расходам (overhead), так как система не выполняет

никаких полезных действий при переключении с одного процесса на другой.

Время переключения контекста зависит от аппаратной поддержки. Выше мы рассматривали необходимые

действия системы в таких случаях – сохранение состояния приостанавливаемого процесса в его PCB и

восстановление состояния возобновляемого процесса из его PCB.

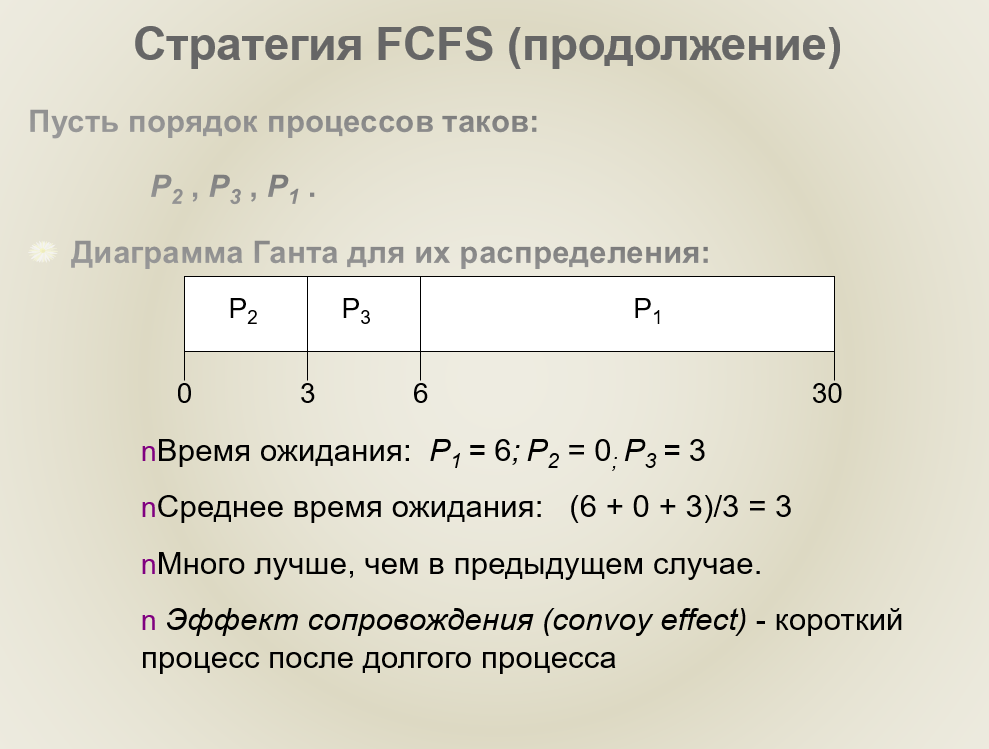
Например, в системе "Эльбрус" контекстное переключение выполнялось всего одной аппаратной командой

- СМСТЕК (сменить стек, т.е. переключиться с одного облегченного процесса на другой). Однако следует

отметить, что такая аппаратная оптимизация была возможна, так как понятие процесса в "Эльбрусе" было

фактически сведено к понятию облегченного процесса (lightweight process).

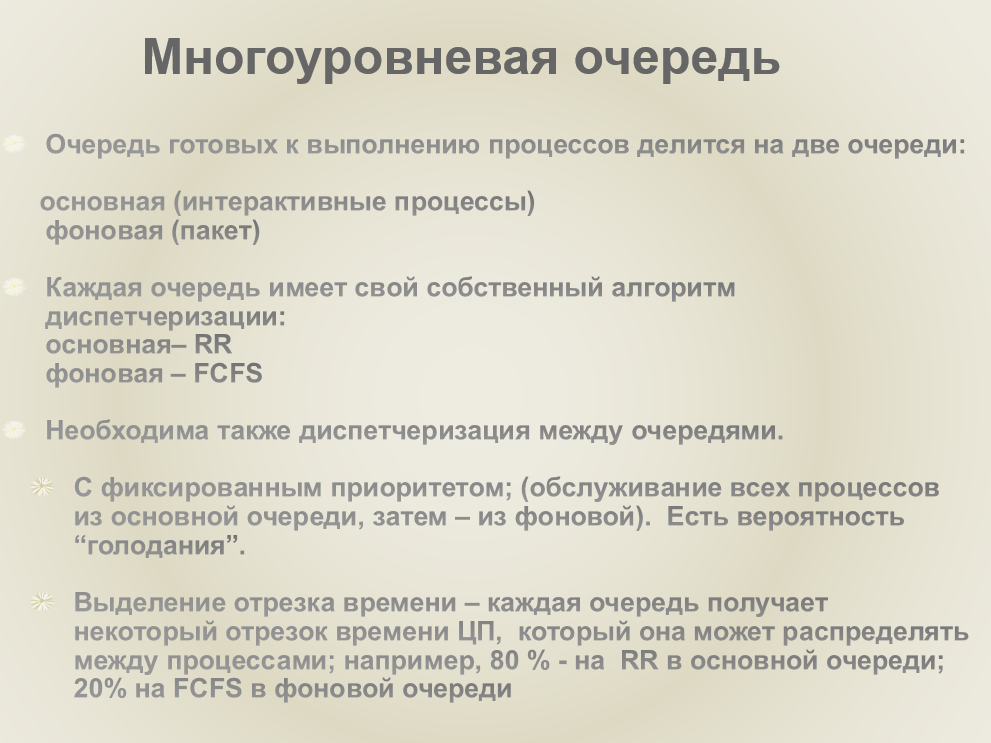


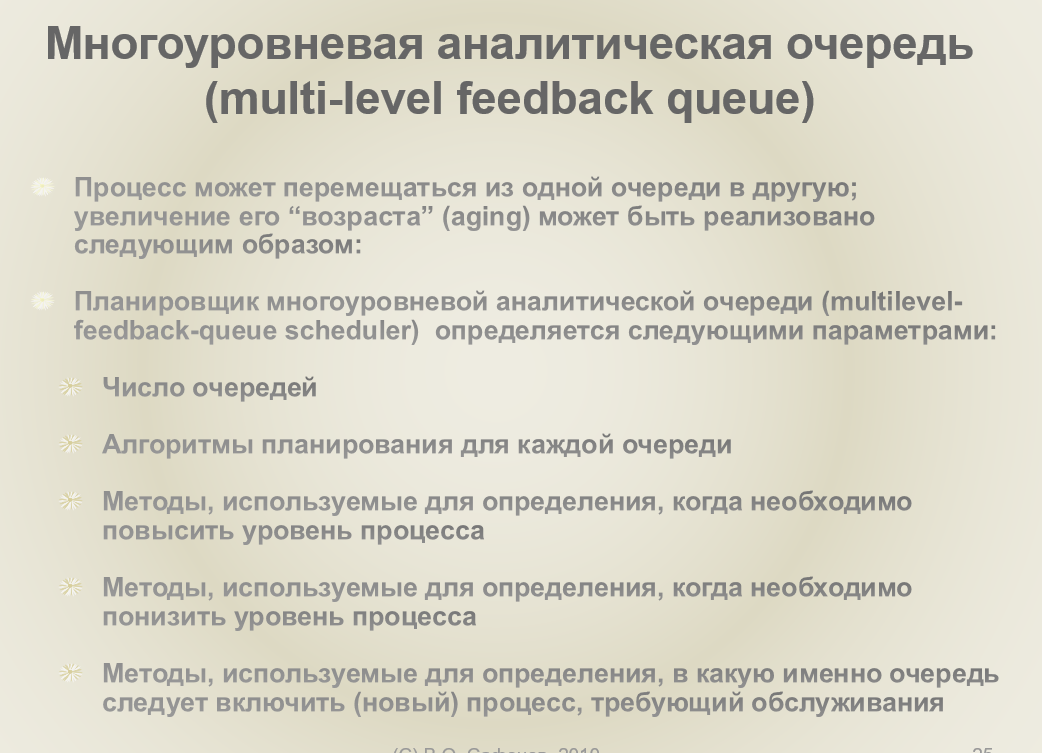
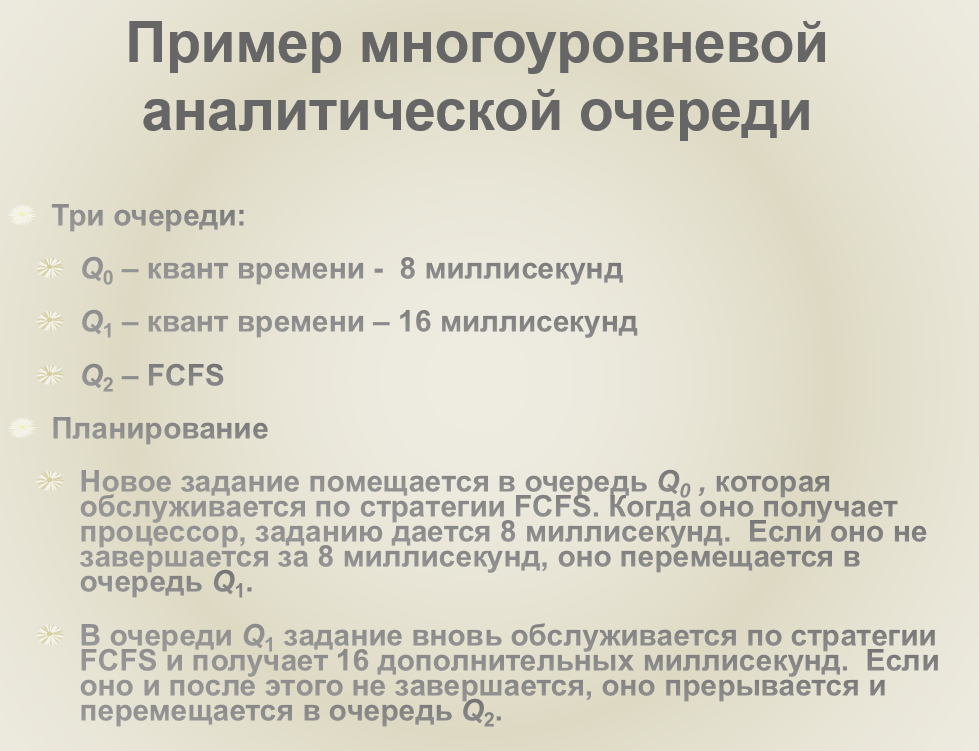
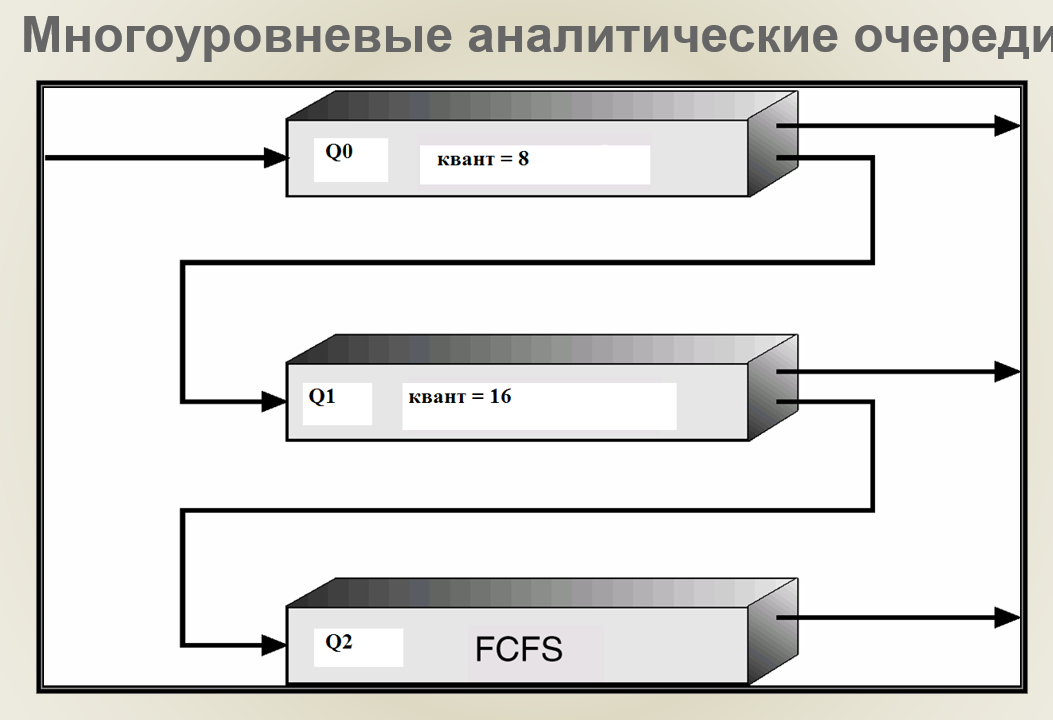






////////////////////////////////////////////////////////////

Диспетчеризация процессора – предоставление всем процессам в системе по очереди в определенном порядке квантов процессорного времени. Главной целью диспетчеризации является максимальная *загрузка процессора*.

Работа любого процесса в системе представляется как последовательность *чередований* фаз активности процессора и активности ввода-вывода. Частота периодов активности процессора обратно пропорциональна их длительности.

*Планировщик* – компонента ОС, планирующая выделение квантов времени процессам по определенной стратегии. Различаются стратегии с прерыванием процессов (когда при вводе нового более короткого или более *приоритетного процесса* в систему текущий процесс прерывается) и без прерывания процессов.

*Диспетчер* – компонента ОС, выполняющая само переключение процессора с одного процесса на другой. Время, которое на это требуется, называется скрытой активностью (*латентностью*) диспетчера и должно быть минимизировано.

Основные критерии диспетчеризации – использование процессора (максимизируется), *пропускная способность* системы (максимизируется), *среднее время* обработки одного процесса (минимизируется), *среднее время* ожидания одним процессом (минимизируется), *среднее время* ответа системы (минимизируется).

Для иллюстрации стратегий диспетчеризации используются *диаграммы Ганта* с изображением временной линейки, имен и периодов активности каждого процесса.

Стратегия диспетчеризации *First*-Come-*First*-Served (*FCFS*) – предоставление ресурсов процессора процессам в порядке их ввода в систему, независимо от их длительности. При этом *время ожидания* может оказаться большим, особенно если первым в систему вводится более длительный процесс (что называется эффектом сопровождения). Ситуация, когда первыми вводятся более короткие процессы, более благоприятна.

Стратегия Shortest-*Job*-*First* (SJF) – предоставление ресурсов процессора более короткому процессу или сопроцессу, оставшееся *время выполнения* которого минимально (Shortest-Remaining-*Time*-*First*, SRTF). Данная стратегия обеспечивает минимальное *среднее время* ожидания процессов.

Метод экспоненциального усреднения позволяет вычислить предсказываемую длину следующего периода активности по фактическим и предсказанным длинам предыдущих периодов активности.

Диспетчеризация по приоритетам предоставляет первым ресурсы процессора более высокоприоритетному процессу. Чтобы избежать ситуации "голодания", ОС постепенно повышает *приоритеты процессов*, длительное время находящихся в системе.

Стратегия *Round Robin* (RR) предоставляет всем процессам по очереди одинаковые кванты времени процессора. Квант времени не должен быть слишком мал, иначе накладные *расходы* на переключение процессов оказываются сравнимыми с полезным временем процессора. Стратегия RR обеспечивает лучшее время ответа, чем SJF, но худшее время оборота.

Число *переключений контекста* с процесса на процесс возрастает с уменьшением выделяемого кванта времени. Время оборота зависит от кванта времени более сложным образом.

Для обработки процессов различных классов и приоритетов (например, пакетных и интерактивных) ОС создает многоуровневые аналитические очереди процессов, каждая из которых обслуживается по различным стратегиям и (или) предоставляет процессам кванты времени различного размера. Процесс при необходимости может быть переведен из одной очереди в другую.

При планировании загрузки многопроцессорных систем учитывается их *симметричность* или асимметричность. Планирование их загрузки гораздо более сложно. В асимметричных системах не требуется синхронизировать процессы по системным структурам данных, так как они доступны процессу только на одном процессоре.

Для систем реального времени наиболее важным является предоставление наивысших приоритетов критическим процессам реального времени, решающим основную задачу системы.

В ОС Solaris и *Windows 2000* выделяются процессы нескольких классов, для которых, соответственно, выделяются различные приоритеты. В системе Solaris для каждого класса процессов имеется свой *планировщик*.

# Планировщик. Стратегия без прерывания процессов. Диаграмма Ганта(СХЕМЫ И ЕСТЬ ДИАГРАММА)

**Стратегия Shortest Job First (SJF, обслуживание самого короткого задания первым)**– стратегия диспетчеризации процессора, при которой *процессор* предоставляется в первую *очередь* наиболее короткому процессу из имеющихся в системе.

В данном случае с каждым процессом связывается *длина* его очередного периода активности. Эта *длина* используется для того, чтобы первым обслужить самый короткий процесс .

Возможны две схемы применения данной стратегии:

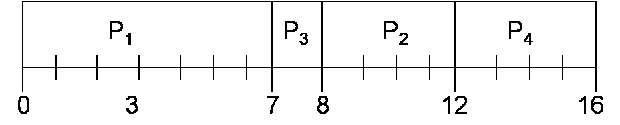
1. **Без прерывания процессов**– пока процессу предоставляется процессор, он не может быть прерван, пока не истечет его квант времени.
2. **С прерыванием процессов**– если приходит новый процесс, время активности которого меньше, чем оставшееся время *активного процесса*, - прервать *активный процесс*. Эта схема известна под названием **Shortest-Remaining-Time-First (SRTF)**.

Нетрудно видеть, что стратегия SJF оптимальна, в том смысле, что она обеспечивает минимальное *среднее время* ожидания для заданного набора процессов.

Рассмотрим пример применения стратегии SJF без прерывания процессов. Пусть набор процессов, времен их появления в системе и времен их активности, следующие:

|  |  |  |
| --- | --- | --- |
| Процесс | Время появления | Время активности |
| P1 | 0.0 | 7 |
| P2 | 2.0 | 4 |
| P3 | 4.0 | 1 |
| P4 | 5.0 | 4 |

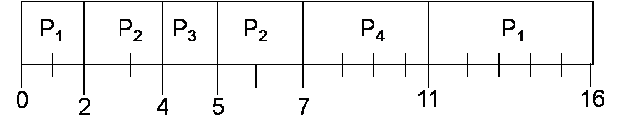
Схема их диспетчеризации по стратегии SJF без прерывания процессов приведена на [рис. 11.5](https://intuit.ru/studies/courses/641/497/lecture/11286?page=2#image.11.5).



**Рис. 11.5.**Схема диспетчеризации процессов по стратегии SJF без прерывания.

В данном случае *среднее время* ожидания = (0 + 6 + 3 + 7)/4 = 4.

Теперь применим к тем же процессам стратегию SJF с прерыванием и проанализируем, как изменится *среднее время* ожидания. Результат применения стратегии изображен на [рис. 11.6](https://intuit.ru/studies/courses/641/497/lecture/11286?page=2#image.11.6).



**Рис. 11.6.**Схема диспетчеризации процессов по стратегии SJF с прерываниями.

В данном случае принцип прерывания процесса в момент поступления в систему более короткого процесса применяется несколько раз:

* в момент 2 прерывается процесс 1 и начинает исполняться более короткий процесс 2;
* в момент 4 прерывается процесс 2 и начинает исполняться более короткий процесс 3.

Из диаграммы видно, что, вследствие применения принципа прерывания процессов, периоды непрерывного выполнения процесса на процессоре могут быть не смежными и перемежаться с периодами выполнения других процессов.

В данном случае *среднее время* ожидания = (9 + 1 + 0 +2)/4 = 3, т.е. оно, как и следовало предполагать, оказалось меньше, чем без применения принципа прерывания процессов.

# Планировщик. Стратегия с прерыванием процессов. Диаграмма Ганта

**Стратегия Shortest Job First (SJF, обслуживание самого короткого задания первым)**– стратегия диспетчеризации процессора, при которой *процессор* предоставляется в первую *очередь* наиболее короткому процессу из имеющихся в системе.

В данном случае с каждым процессом связывается *длина* его очередного периода активности. Эта *длина* используется для того, чтобы первым обслужить самый короткий процесс .

Возможны две схемы применения данной стратегии:

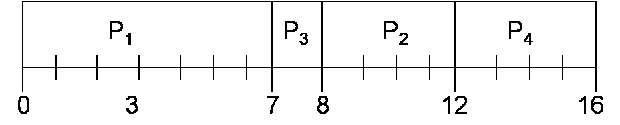
1. **Без прерывания процессов**– пока процессу предоставляется процессор, он не может быть прерван, пока не истечет его квант времени.
2. **С прерыванием процессов**– если приходит новый процесс, время активности которого меньше, чем оставшееся время *активного процесса*, - прервать *активный процесс*. Эта схема известна под названием **Shortest-Remaining-Time-First (SRTF)**.

Нетрудно видеть, что стратегия SJF оптимальна, в том смысле, что она обеспечивает минимальное *среднее время* ожидания для заданного набора процессов.

Рассмотрим пример применения стратегии SJF без прерывания процессов. Пусть набор процессов, времен их появления в системе и времен их активности, следующие:

|  |  |  |
| --- | --- | --- |
| Процесс | Время появления | Время активности |
| P1 | 0.0 | 7 |
| P2 | 2.0 | 4 |
| P3 | 4.0 | 1 |
| P4 | 5.0 | 4 |

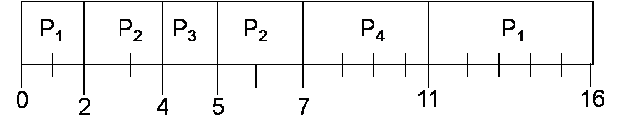
Схема их диспетчеризации по стратегии SJF без прерывания процессов приведена на [рис. 11.5](https://intuit.ru/studies/courses/641/497/lecture/11286?page=2#image.11.5).



**Рис. 11.5.**Схема диспетчеризации процессов по стратегии SJF без прерывания.

В данном случае *среднее время* ожидания = (0 + 6 + 3 + 7)/4 = 4.

Теперь применим к тем же процессам стратегию SJF с прерыванием и проанализируем, как изменится *среднее время* ожидания. Результат применения стратегии изображен на [рис. 11.6](https://intuit.ru/studies/courses/641/497/lecture/11286?page=2#image.11.6).



**Рис. 11.6.**Схема диспетчеризации процессов по стратегии SJF с прерываниями.

В данном случае принцип прерывания процесса в момент поступления в систему более короткого процесса применяется несколько раз:

* в момент 2 прерывается процесс 1 и начинает исполняться более короткий процесс 2;
* в момент 4 прерывается процесс 2 и начинает исполняться более короткий процесс 3.

Из диаграммы видно, что, вследствие применения принципа прерывания процессов, периоды непрерывного выполнения процесса на процессоре могут быть не смежными и перемежаться с периодами выполнения других процессов.

В данном случае *среднее время* ожидания = (9 + 1 + 0 +2)/4 = 3, т.е. оно, как и следовало предполагать, оказалось меньше, чем без применения принципа прерывания процессов.

# Планировщик. стратегии FCFS и каковы ее недостатки



Процессы ставятся в очередь по мере поступления.

Преимущества: Простата Справедливость (как в очереди покупателей, кто последний пришел, тот оказался в конце очереди)

Недостатки: Процесс, ограниченный возможностями процессора может затормозить более быстрые процессы, ограниченные устройствами ввода/вывода.

Прямой порядок выполнения (оказался неэффективным — процессы долго находятся в состоянии готовности).

Прямой порядок выполнения (более эффективный чем первый).

Недостатки: Если частые переключения (квант - 4мс, а время переключения равно 1мс), то происходит уменьшение производительности.

Если редкие переключения (квант - 100мс, а время переключения равно 1мс), то происходит увеличение времени ответа на запрос.

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно RR. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд. Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины кванта времени, равной 1. Время ожидания для процессаp составит 5 единиц времени, для процесса p1 – тоже 5 единиц, для процесса p2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

# Планировщик. стратегии SJF (и SRTF) и оптимальность по какому критерию она обеспечивает

**Стратегия Shortest Job First (SJF, обслуживание самого короткого задания первым)**– стратегия диспетчеризации процессора, при которой *процессор* предоставляется в первую *очередь* наиболее короткому процессу из имеющихся в системе.

В данном случае с каждым процессом связывается *длина* его очередного периода активности. Эта *длина* используется для того, чтобы первым обслужить самый короткий процесс .

Возможны две схемы применения данной стратегии:

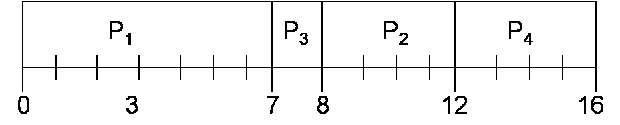
1. **Без прерывания процессов**– пока процессу предоставляется процессор, он не может быть прерван, пока не истечет его квант времени.
2. **С прерыванием процессов**– если приходит новый процесс, время активности которого меньше, чем оставшееся время *активного процесса*, - прервать *активный процесс*. Эта схема известна под названием **Shortest-Remaining-Time-First (SRTF)**.

Нетрудно видеть, что стратегия SJF оптимальна, в том смысле, что она обеспечивает минимальное *среднее время* ожидания для заданного набора процессов.

Рассмотрим пример применения стратегии SJF без прерывания процессов. Пусть набор процессов, времен их появления в системе и времен их активности, следующие:

|  |  |  |
| --- | --- | --- |
| Процесс | Время появления | Время активности |
| P1 | 0.0 | 7 |
| P2 | 2.0 | 4 |
| P3 | 4.0 | 1 |
| P4 | 5.0 | 4 |

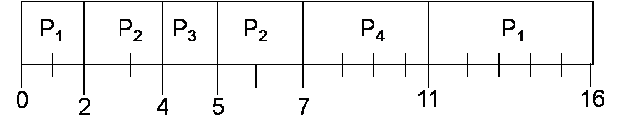
Схема их диспетчеризации по стратегии SJF без прерывания процессов приведена на [рис. 11.5](https://intuit.ru/studies/courses/641/497/lecture/11286?page=2#image.11.5).



**Рис. 11.5.**Схема диспетчеризации процессов по стратегии SJF без прерывания.

В данном случае *среднее время* ожидания = (0 + 6 + 3 + 7)/4 = 4.

Теперь применим к тем же процессам стратегию SJF с прерыванием и проанализируем, как изменится *среднее время* ожидания. Результат применения стратегии изображен на [рис. 11.6](https://intuit.ru/studies/courses/641/497/lecture/11286?page=2#image.11.6).



**Рис. 11.6.**Схема диспетчеризации процессов по стратегии SJF с прерываниями.

В данном случае принцип прерывания процесса в момент поступления в систему более короткого процесса применяется несколько раз:

* в момент 2 прерывается процесс 1 и начинает исполняться более короткий процесс 2;
* в момент 4 прерывается процесс 2 и начинает исполняться более короткий процесс 3.

Из диаграммы видно, что, вследствие применения принципа прерывания процессов, периоды непрерывного выполнения процесса на процессоре могут быть не смежными и перемежаться с периодами выполнения других процессов.

В данном случае *среднее время* ожидания = (9 + 1 + 0 +2)/4 = 3, т.е. оно, как и следовало предполагать, оказалось меньше, чем без применения принципа прерывания процессов.

Из диаграммы видно, что, вследствие применения принципа прерывания процессов, периоды

непрерывного выполнения процесса на процессоре могут быть не смежными и перемежаться с периодами выполнения других процессов.

В данном случае среднее время ожидания = (9 + 1 + 0 +2)/4 = 3, т.е. оно, как и следовало предполагать, оказалось меньше, чем без применения принципа прерывания процессов

Предсказание длины следующего периода активности

Попытаемся теперь предложить и применить формулы для предсказания следующего периода активности

процесса. Подобные оценки помогли бы разработчикам ОС реализовать оптимальную стратегию

диспетчеризации. Используем уже известные фактические длины предыдущих периодов активности и

принцип экспоненциального усреднения. Пусть:

tn – фактическая длина n- го периода активности процесса;

– предсказанная длина n- го периода активности процесса.

Будем искать значение для предсказания следующего периода активности процесса как

следующую линейную комбинацию tn и :

.

где – число между 0 и 1 . Коэффициент характеризует, в какой степени при предсказании

учитывается недавняя история вычислений.

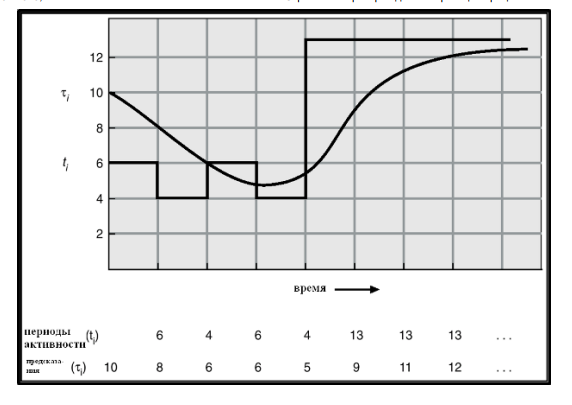
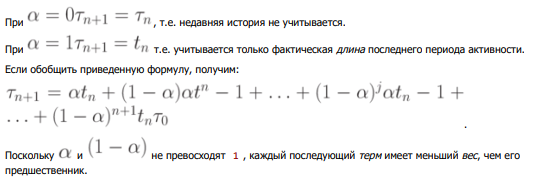
Пример предсказания следующего периода активности по приведенной формуле приведен на рис. 11.7. .

Рис. 11.7. Пример предсказания следующего периода активности.



# Планировщик. Диспетчер. Диспетчеризации по приоритетам

* Выбирает один из нескольких процессов, загруженных в память и готовых к выполнению, и выделяет процессор для одного из них.
* Решения по диспетчеризации могут быть приняты, когда процесс:

1. Переключается из состояния выполнения в состояние ожидания.

2. Переключается из состояния выполнения в состояние готовности к выполнению.

3. Переключается из состояния ожидания в состояние готовности.

4. Завершается.

* Диспетчеризация типов 1 и 4 – *без прерывания процесса*

*(non-preemptive)*.

* В остальных случаях – *с прерыванием процесса (preemptive).*

**Собственно диспетчер**

* Модуль диспетчера предоставляет процессор тому процессу, который был выбран планировщиком, то есть:
  + Переключает контекст
    - Переключает процессор в пользовательский режим
    - Выполняет переход по соответствующему адресу в пользовательскую программу для ее рестарта
* Скрытая активность диспетчера (*dispatch latency)* – время, требуемое для диспетчера, чтобы остановить один процесс и стартовать другой.
* **Критерии диспетчеризации**
* Использование процессора – поддержание его в режиме занятости, насколько это возможно
* Пропускная способность (throughput) – число процессов, завершающих свое выполнение за единицу времени
* Время обработки (turnaround time) – время, необходимое для исполнения какого-либо процесса
* Время ожидания (waiting time)– время, которое процесс ждет в очереди процессов, готовых к выполнению
* Время ответа (response time) – время, требуемое от момента первого запроса до первого ответа (для среды разделения времени)

Диспетчеризация по приоритетам

При данной стратегии с каждым процессом связывается его приоритет (целое число). Процессор

выделяется процессу с наивысшим приоритетом (будем считать, что меньшее число означает более

высокий приоритет процесса, т.е. наиболее высокий приоритет процесса равен 1).

Данная стратегия, как и предыдущая, имеет варианты с прерыванием и без прерывания.

Более того, стратегию SJF можно рассматривать как диспетчеризацию по приоритетам, в которой

приоритетом является очередное время активности.

При диспетчеризации по приоритетам возникает проблема "голодания" (starvation) - ситуации, когда

процессы с низким приоритетом могут никогда не исполниться и бесконечно ждать.

Традиционным способом решение данной проблемы в операционных системах является учет возраста

процесса ( aging ): c течением времени приоритет процесса повышается системой.

# Планировщик. Проблема голодания процессов и каково ее решение в ОС

**Голодание** (starvation) - ситуация в системе, когда процессы с низким приоритетом длительное время ждут и не получают квантов времени процессора.

Голодание может быть предотвращено с помощью надлежащего алгоритма планирования с приоритетной очередью, который на самом деле также использует метод старения - метод планирования, который добавляет фактор старения к приоритету каждого запроса, что означает, что он увеличивает уровень приоритета низкоприоритетных процессов, которые ожидали в течение долгого времени. Кроме того, предоставление дополнительных ресурсов программам должно избегать продолжения перегруженности ресурсов. Чтобы система не зашла в тупик, процессам необходимо запретить доступ к одному или нескольким ресурсам, одновременно ожидая других, и только один процесс должен иметь доступ к ресурсу за раз.

///////////////////////////////////////////////////////////

Проблема полной недоступности ресурсов определённому потоку в связи с потреблением их другими потоками. Может проявляется при некорректном планирования выполнения (например различие в приоритетах на определённых системах), либо более частый случай - ошибки при проектировании и реализации. Если у одного потока будет выставлен очень высокий приоритет, а у другого низкий, то первый может потреблять существенную все вычислительные ресурсы, в то время как второй не получит их вовсе. К счастью, такое поведение практически невозможно на большинстве платформ поддерживаемых Java. В то время как ошибки при проектировании и реализации приводящие к проблеме голодания вполне реальны. Например ошибка некорректной обработки условия выхода из цикла выполняющегося при захваченной блокировке, которую ожидает другой поток.

Детектирование проблемы голодания не столь прямолинейно как в случае с взаимной блокировкой. Если ситуация подобна примеру, то можно легко вычислить не ожидающий поток с бесконечным циклом любой утилитой отображающей состояние процессов и потоков в операционной системе. Далее получить дамп потоков и определить место возникновения проблемы. Но если в такой ситуации внутри цикла поток будет блокироваться в ожидании события или просто отдавать рекомендацию планировщику (Thread.yield, Thread.sleep) уступить текущий квант, то определить её будет сложнее. При подобных условиях могут пригодиться как ручное наблюдение за стеками выполнения и состояниями потоков во времени, так и автоматическое профилирование приложения (может понадобиться длительное время сбора статистики).

# Стратегии RR, оптимальность по какому критерию она обеспечивает и по какому критерию она хуже, чем SJF

### Предсказание длины следующего периода активности

Попытаемся теперь предложить и применить формулы для предсказания следующего периода *активности процесса*. Подобные оценки помогли бы разработчикам ОС реализовать *оптимальную стратегию* диспетчеризации. Используем уже известные фактические длины предыдущих периодов активности и принцип экспоненциального усреднения. Пусть:

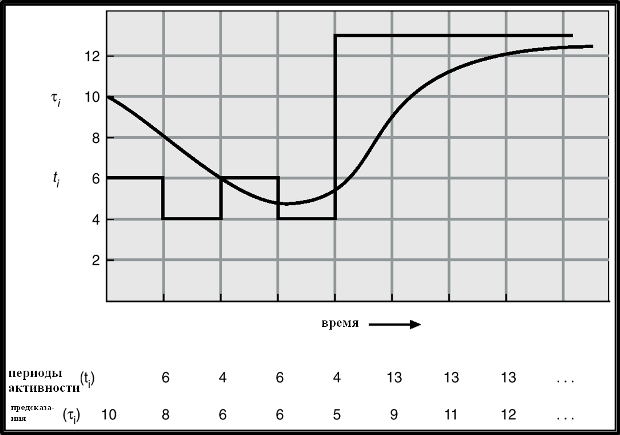
* tn – фактическая длина n- го периода *активности процесса*;
* \tau _{n} – предсказанная длина n- го периода *активности процесса*.

Будем искать *значение* \tau _{n+1} для предсказания следующего периода *активности процесса* как следующую *линейную комбинацию* tn и \tau _{n}:

\tau _{n+1} = \alpha  t_{n} + (1 – \alpha ) \tau _{n}.

где \alpha – число между 0 и 1. Коэффициент \alpha характеризует, в какой степени при предсказании учитывается недавняя история вычислений.

Пример предсказания следующего периода активности по приведенной формуле приведен на [рис. 11.7](https://intuit.ru/studies/courses/641/497/lecture/11286?page=3#image.11.7).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/11/files/11_7.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/11/files/11_7.png)  
**Рис. 11.7.**Пример предсказания следующего периода активности.

При \alpha  =0 \tau _{n+1} = \tau _{n}, т.е. недавняя история не учитывается.

При \alpha  =1 \tau _{n+1} = t_{n} т.е. учитывается только фактическая *длина* последнего периода активности.

Если обобщить приведенную формулу, получим:

\tau _{n+1} = \alpha  t_{n}+(1 - \alpha ) \alpha  t^{n} -1 + … +(1 - \alpha  )^{j} \alpha  t_{n} -1 + … +(1 - \alpha  )^{n+1} t_{n} \tau _{0}.

Поскольку \alpha и (1 - \alpha ) не превосходят 1, каждый последующий *терм* имеет меньший *вес*, чем его предшественник.

### Диспетчеризация по приоритетам

При данной стратегии с каждым процессом связывается его приоритет (*целое число*). *Процессор* выделяется процессу с наивысшим приоритетом (будем считать, что меньшее число означает более высокий *приоритет процесса*, т.е. наиболее высокий *приоритет процесса* равен 1).

Данная стратегия, как и предыдущая, имеет варианты с прерыванием и без прерывания.

Более того, стратегию SJF можно рассматривать как диспетчеризацию по приоритетам, в которой приоритетом является очередное время активности.

При диспетчеризации по приоритетам возникает проблема **"голодания" (starvation)**- ситуации, когда процессы с низким приоритетом могут никогда не исполниться и бесконечно ждать.

Традиционным способом решение данной проблемы в операционных системах является учет **возраста**процесса ( **aging** ): c течением времени *приоритет процесса* повышается системой.

### Стратегия Round Robin (RR)

**Стратегия Round Robin (RR, круговая система)**– это предоставление всем процессам по очереди одинаковых квантов времени. Название стратегии происходит от названия популярной в США карточной игры.

При данной стратегии каждый процесс получает небольшой *квант процессорного времени*, обычно – 10-100 миллисекунд. После того, как это время закончено, процесс прерывается и помещается в конец очереди готовых процессов.

Если всего имеется n процессов в очереди готовых к выполнению, и квант времени равен q, то каждый процесс получает 1/ n процессорного времени порциями самое большее по q единиц за один раз. Ни один процесс не ждет больше, чем (n-1) q единиц времени.

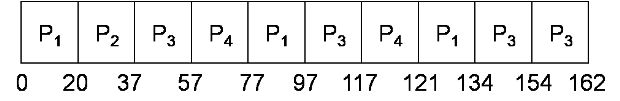
*Производительность* данной стратегии зависит от коэффициента q:

* если q велико, то стратегия фактически эквивалентна стратегии *FCFS*;
* если q мало, то q должно быть большим, чем время контекстного переключения, иначе слишком велики окажутся накладные расходы на переключения с одного процесса на другой.

Рассмотрим пример применения стратегии RR. Пусть в системе имеются следующие процессы со следующими временами активности:

|  |  |
| --- | --- |
| Процесc | Время активности |
| P1 | 53 |
| P2 | 17 |
| P3 | 68 |
| P4 | 24 |

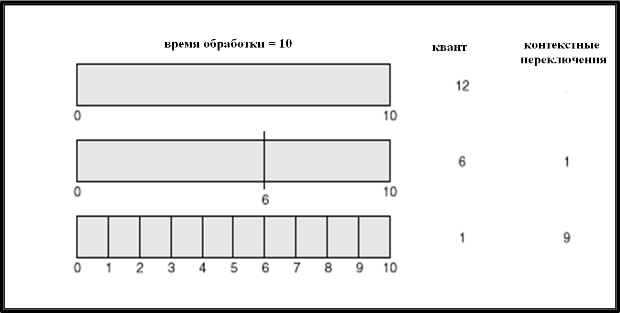
Схема диспетчеризации процессора по стратегии RR с квантом времени q = 20 приведена на [рис. 11.8](https://intuit.ru/studies/courses/641/497/lecture/11286?page=3#image.11.8).



**Рис. 11.8.**Пример применения стратегии RR (q = 20).

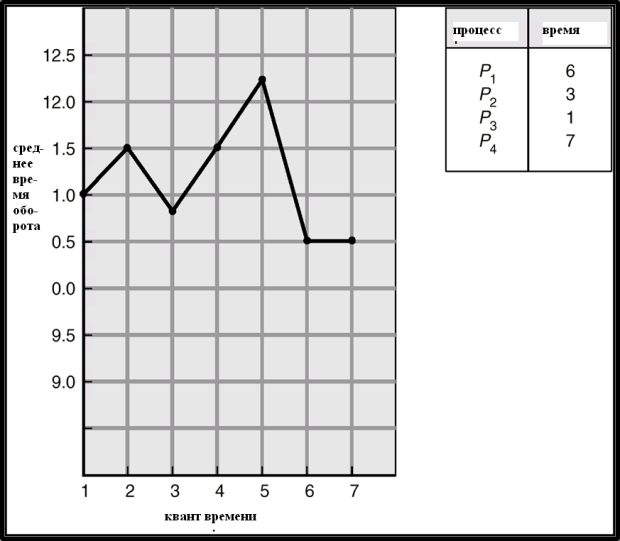
Обычно стратегия RR имеет худшее время оборота, чем SJF (так как каждый процесс должен ждать до предоставления следующего кванта времени, пока кванты времени будут предоставлены всем другим процессам), но лучшее время ответа.

На [рис. 11.9](https://intuit.ru/studies/courses/641/497/lecture/11286?page=3#image.11.9) иллюстрируется зависимость числа контекстных переключений от кванта времени: чем меньше квант, тем больше число *переключений контекста*.

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/11/files/11_9.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/11/files/11_9.png)  
**Рис. 11.9.**Квант времени процессора и время переключения контекста.

На [рис. 11.10](https://intuit.ru/studies/courses/641/497/lecture/11286?page=3#image.11.10) приведен пример зависимости времени оборота от кванта времени. В данном случае зависимость носит более сложный характер.

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/11/files/11_10.png)

### Многоуровневая очередь

Поскольку процессы в системе могут иметь различную специфику (например, пакетные и интерактивные), на практике в операционных системах *очередь* готовых к выполнению процессов делится на две очереди:

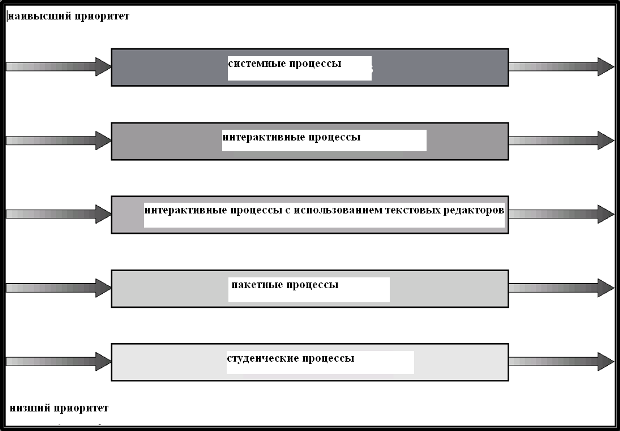
* **основная (интерактивные процессы)**
* **фоновая (пакетные процессы)**.

Каждая *очередь* имеет свой собственный *алгоритм* диспетчеризации: основная –RR, фоновая – *FCFS*.

При данной *смешанной стратегии* необходима также диспетчеризация между очередями, т.е. стратегия выбора процессов из той или иной очереди. Различаются следующие виды диспетчеризации между очередями:

* **С фиксированным приоритетом**- обслуживание всех процессов из основной очереди, затем – из фоновой. При этом имеется вероятность "голодания".
* **Выделение отрезка времени**– каждая очередь получает некоторый отрезок времени ЦП, который она может распределять между процессами; например, 80% - на RR в основной очереди; 20% на *FCFS* в фоновой очереди.

На [рис. 11.11](https://intuit.ru/studies/courses/641/497/lecture/11286?page=3#image.11.11) приведен реалистичный пример структуры *многоуровневой очереди* для диспетчеризации процессов. Наивысший приоритет имеют системные процессы, далее – интерактивные, еще меньший – интерактивные с вызовами текстовых редакторов (они занимают значительно больше времени из-за медленной работы пользователей); затем следуют пакетные и, наконец, студенческие процессы. Такова реальная ситуация, хотя *автор* и не считает справедливым "дискриминацию" студенческих процессов: возможны ситуации, когда именно им следует отдавать наивысший приоритет после системных – например, незадолго до защит дипломных *работ*.

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/11/files/11_11.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/11/files/11_11.png)  
**Рис. 11.11**

# В чем заключается задача управления памятью? входная очередь заданий.

Введение

Управление памятью, наряду с управлением процессами и ресурсами, - одна из наиболее важных функций операционной системы. Задача ОС заключается в том, чтобы размещать в памяти пользовательские процессы, их данные, обслуживать запросы процессов на области памяти заданных размеров. В данной лекции рассмотрены базовые понятия и механизмы, связанные с управлением памятью, в частности, этапы загрузки программ в память и их связывание с адресами в памяти. Две основные стратегии управления памятью – страничная организация и сегментная организация – рассмотрены в следующих двух лекциях "Страничная организация памяти" и "Сегментная организация памяти".

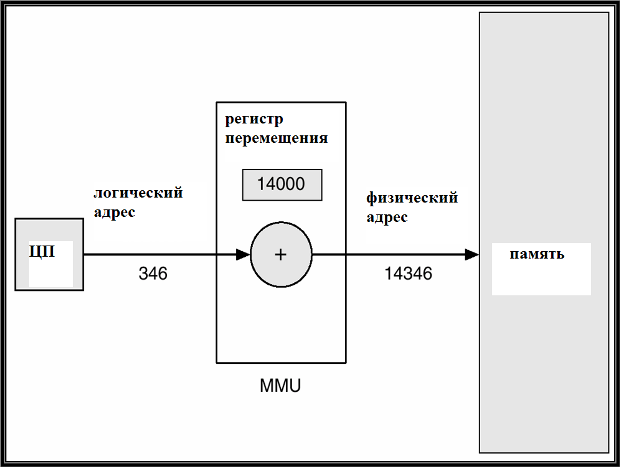
### Устройство управления памятью

Как уже отмечалось во вводной лекции, **устройство управления памятью (** **Memory Management Unit – MMU** **)**– это один из модулей аппаратуры, отвечающий за адресацию памяти и связанный с процессором и другими устройствами системной шиной. С точки зрения поддержки описанных концепций адресации, *устройство управления памятью* – это аппаратура, преобразующая *логический адрес* (полученный по *общей шине* от процессора) в физический (реальный *адрес* в памяти, по которому и происходит обращение).

Аппаратура *MMU* использует *значение* **регистра перемещения**, содержащего *адрес* начала области памяти, выделенной ОС для программы пользователя. *MMU* добавляет *значение* регистра перемещения к (логическому) адресу, сгенерированному пользовательской программой, получая в результате *физический адрес*.

*Программа* пользователя работает только с *логическими адресами* и не "видит" *физических адресов*.

Схема адресации и преобразования *логического адреса* в физический с использованием регистра перемещения изображена на [рис. 15.2](https://intuit.ru/studies/courses/641/497/lecture/11294?page=1#image.15.2).



**Рис. 15.2.**Адресация с использованием регистра перемещения.

# связывание адресов и на каких этапах обработки программы оно может выполняться

### Связывание программ и данных с адресами в памяти

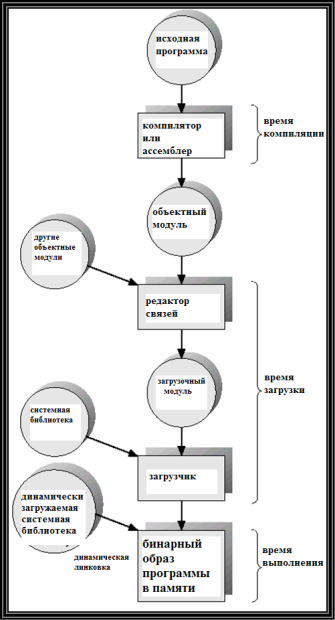
Перед *загрузкой данных* или кода в *память* они должны быть в какой-либо момент связаны с определенными адресами в памяти. *Связывание* может выполняться на разных этапах:

* **Связывание во время компиляции (compile-time)**.Если адрес в памяти априорно известен, компилятором может быть сгенерирован код с абсолютными адресами. При любом изменении размещения программы в памяти должна быть выполнена перекомпиляция. Данный подход более характерен для ранних компьютерных систем с небольшим объемом памяти, либо для обработки и выполнения системных модулей – частей ядра ОС, для которых характерно использование резидентных абсолютных адресов. Для пользовательских программ такой подход неудобен, так как не обеспечивает достаточной гибкости, в частности, возможности без изменений перезагрузить код в другую область памяти.
* **Связывание во время загрузки (load-time)**.Загрузка программы в память – стадия ее обработки системой, предшествующая выполнению программы. Чтобы начальный адрес области памяти, куда загружается программа, можно было менять, и это не привело бы к необходимости изменения кода программы, применяется следующий метод. Генерируется **перемещаемый код (relocatable code)**– код, в котором адресация происходит относительно значения **регистра перемещения (relocation register)**,и адрес в памяти равен сумме значения регистра перемещения и адреса, вычисляемого в команде. Таким образом, при необходимости загрузки кода на другое место в памяти требуется изменить только значение регистра перемещения. Подобный подход широко используется для программ, написанных на традиционных языках программирования.
* **Связывание во время исполнения (runtime)**,или **динамическое (позднее) связывание**.Используется, если процесс во время выполнения может быть перемещен из одного сегмента памяти в другой. Для реализации связывания во время исполнения требуется аппаратная поддержка отображения адресов – например, регистры базы и границы. В большинстве систем для пользовательских программ используется, главным образом, именно связывание во время исполнения.

# этапы обработки проходит программа на пути от исходного кода к двоичному образу в памяти

### Многоэтапная обработка пользовательской программы

Чтобы лучше представлять себе все детали адресации и размещения программы в памяти, рассмотрим общую схему многоэтапной обработки пользовательской программы, используемую в любой ОС. Схема представлена на [рис. 15.1](https://intuit.ru/studies/courses/641/497/lecture/11294?page=1#image.15.1).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/15/files/15_1.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/15/files/15_1.png)  
**Рис. 15.1.**Многоэтапная обработка пользовательской программы.

**Исходный код**программы (в форме текстового файла) на *языке высокого уровня* или на *ассемблере* преобразуется компилятором или *ассемблером* в **объектный модуль**, содержащий бинарные выполняемые машинные команды и **таблицу символов**, определенных и использованных в данном модуле кода. Рассмотренная фаза называется **временем компиляции**.

Однако объектный *модуль* не может непосредственно исполняться, так как он содержит неразрешенные ссылки на внешние модули и их компоненты. Следующая фаза обработки программы – **редактирование связей. Редактор связей (linker)**– *системная программа*, которая получает на вход один или несколько объектных модулей, а на выходе выдает **загрузочный модуль**– двоичный код, образованный кодом нескольких объектных модулей, в котором разрешены все межмодульные ссылки - для каждого символа, внешнего для данного объектного модуля **A**, найден соответствующий символ (процедуры, переменной и т.д.) из другого модуля **B**, на который ссылается *модуль* **A**, и код соответственно откорректирован, т.е. он правильно адресует внешний символ.

Загрузочный *модуль* может быть загружен в *память* для исполнения с помощью еще одной системной программы – **загрузчика (loader)**,который получает на вход загрузочный *модуль* и файлы с бинарными кодами **системных библиотек**,которые использует *программа*. *Загрузчик*, объединяя *код программы* с кодами системных библиотек, создает **бинарный образ программы в памяти**.

Фаза вызова *редактора связей* и *загрузчика* носит общее название **время загрузки**. Во многих ОС функции *редактора связей* и *загрузчика*, с целью экономии времени обработки программы в системе, объединены в одной *системной программе* – **редакторе связей и загрузчике (linker and loader)**.Например, в системе *UNIX* *редактор связей* и *загрузчик* называется **ld (Linker and loaDer)**.Объединенному *загрузчику* и редактору связей на вход передается *список* объектных модулей и *список* библиотек, и в результате он генерирует *исполняемый код*. Фаза редактирования связей и загрузки часто на программистском слэнге называется **линковкой (linking)**.Будем далее использовать именно этот короткий и выразительный термин.

Вот пример последовательности фаз обработки программы в терминах команд системы *UNIX*:

сс –c program.c // Компиляция исходного кода на Си.

// В рабочей директории – объектный модуль program.o

ld program.o mylibrary.a // редактирование связей и загрузка

// В рабочей директории – исполняемый код с именем по умолчанию a.out

a.out // Исполнение программы

// В стандартный вывод (по умолчанию – на консоль)

// выдаются результаты программы

В примере предполагается, что в файле **program.c**хранится исходный *код программы* на *Си*, которая использует библиотечные функции из библиотеки **mylibrary.a**. Отметим соглашения в системе *UNIX* о расширениях имен файлов: **.c –**исходный код на *Си*, **.o**– объектный *модуль*, **.a**– *бинарный файл* **статически линкуемой библиотеки**(аббревиатура от термина **archive** ). *Исполняемый код* (executable) в *UNIX* не имеет стандартного расширения имени, но имеет *полное имя* по умолчанию – несколько архаичное имя **a.out**(аббревиатура от **asembler output** ).

В *Windows* соглашения о расширениях имен файлов несколько иные: **.obj**– объектный *модуль*, **.exe**– *исполняемый код*, **.lib**– статически линкуемая библиотека.

Отличить объектный *модуль* от загрузочного очень просто: они сильно отличаются по своему размеру. Объясняется это тем, что в загрузочном модуле присутствует полностью или частично код статически линкуемых библиотек, а также гораздо больше, чем у объектного модуля, *таблица символов* – она содержит все символы библиотек и других объектных модулей, слинкованных в единую исполняемую программу.

На этапе выполнения, при первом обращении к ним из программы, в *память* загружаются **динамически линкуемые библиотеки (dymanically linked libraries)**.Данная разновидность библиотек, реализованная во всех современных ОС, позволяет сэкономить *память*, занимаемую образом *исполняемого кода*, который при статической линковке с библиотеками оказывается очень велик. Подробнее об этом – позже в данной лекции.

### Логическое и физическое адресное пространство

Концепция **логического адресного пространства**, связанного с соответствующим физическим адресным пространством, является одной из основных для *управления памятью*.

**Логическим адресом**называется *адрес*, генерируемый процессором при выполнении машинной команды.

**Физический адрес –**это реальный *адрес* в памяти, который "видит" и "понимает" **устройство управления памятью (Memory Management Unit – MMU)**.

Логические адреса совпадают с физическими при связывании адресов во *время компиляции* или во время загрузки (т.е. **до**исполнения программы). Однако при связывании адресов во *время выполнения* логические адреса отличаются от физических. Далее рассмотрим этот вопрос подробнее.

# Адресация с использованием регистра перемещения

**Аппаратура**

Вопросы функционального построения ОС очень тесно связаны с архитектурой аппаратных средств и имеют важное значение с точки зрения работы операционных систем.

**Иерархия памяти**

Современные вычислительные машины содержат несколько видов памяти, в том числе основную (первичную, оперативную), внешнюю (вторичную, массовую) и кэш-память. В основной памяти должны размещаться команды и данные, к которым будет обращаться работающая программа. Внешняя память — это магнитные диски, предназначенные для хранения информации, которая со временем будет записана в основную память. Кэш-память — это буферная память очень высокого быстродействия, предназначенная для повышения скорости выполнения работающих программ; для программ пользователя эта память, как правило, «прозрачна». В вычислительных машинах, оснащенных кэш-памятью, текущая часть программы помещается в кэшпамять, что позволяет выполнять ее гораздо быстрее, чем если бы она находилась в основной памяти. Все эти виды памяти создают единую иерархию памяти; переход по уровням этой иерархии от кэш-памяти к основной и затем к внешней памяти сопровождается уменьшением стоимости и скорости и увеличением емкости памяти. Память, как правило, разделяется на байты (символы) или слова (состоящие из постоянного количества байтов). Каждая ячейка памяти имеет свой адрес; множество адресов, доступных программе, называется адресным пространством.

**Расслоение памяти**

Метод расслоения памяти (интерливинг) применяется для увеличения скорости доступа к основной (оперативной) памяти. В обычном случае во время обращения к какой-то одной из ячеек модуля основной памяти никакие другие обращения к памяти производиться не могут. При расслоении памяти соседние по адресам ячейки размещаются в различных модулях памяти, так что появляется возможность производить несколько обращений одновременно. Например, при расслоении на два направления ячейки с нечетными адресами оказываются в одном модуле памяти, а с четными — в другом. При простых последовательных обращениях к основной памяти ячейки выбираются поочередно. Таким образом, расслоение памяти позволяет обращаться сразу к нескольким ячейкам, поскольку они относятся к различным модулям памяти.

**Регистр перемещения**

Регистр перемещения обеспечивает возможность динамического перемещения программ в памяти. В регистр перемещения заносится базовый адрес программы, хранящейся в основной памяти. Содержимое регистра перемещения прибавляется к каждому указанному в выполняемой программе адресу. Благодаря этому пользователь может писать программу так, как если бы она начиналась с нулевой ячейки памяти. Во время выполнения программы все исполнительные адреса обращений формируются с использованием регистра перемещения — и благодаря этому программа может реально размещаться в памяти совсем не в тех местах, которые она должна была бы занимать согласно адресам, указанным при трансляции.

//////////////

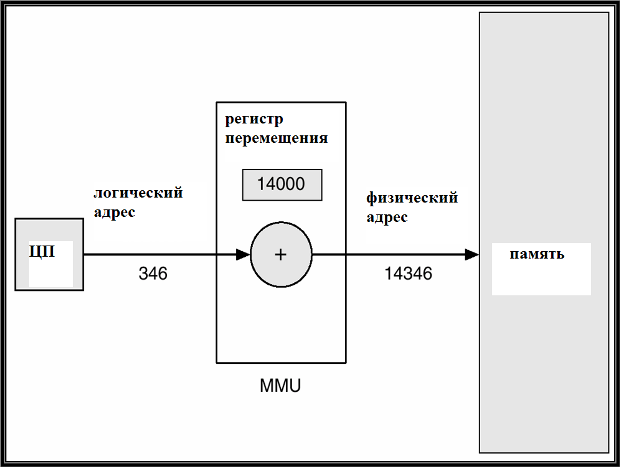
### Устройство управления памятью

Как уже отмечалось во вводной лекции, **устройство управления памятью (** **Memory Management Unit – MMU** **)**– это один из модулей аппаратуры, отвечающий за адресацию памяти и связанный с процессором и другими устройствами системной шиной. С точки зрения поддержки описанных концепций адресации, *устройство управления памятью* – это аппаратура, преобразующая *логический адрес* (полученный по *общей шине* от процессора) в физический (реальный *адрес* в памяти, по которому и происходит обращение).

Аппаратура *MMU* использует *значение* **регистра перемещения**, содержащего *адрес* начала области памяти, выделенной ОС для программы пользователя. *MMU* добавляет *значение* регистра перемещения к (логическому) адресу, сгенерированному пользовательской программой, получая в результате *физический адрес*.

*Программа* пользователя работает только с *логическими адресами* и не "видит" *физических адресов*.

Схема адресации и преобразования *логического адреса* в физический с использованием регистра перемещения изображена на [рис. 15.2](https://intuit.ru/studies/courses/641/497/lecture/11294?page=1#image.15.2).



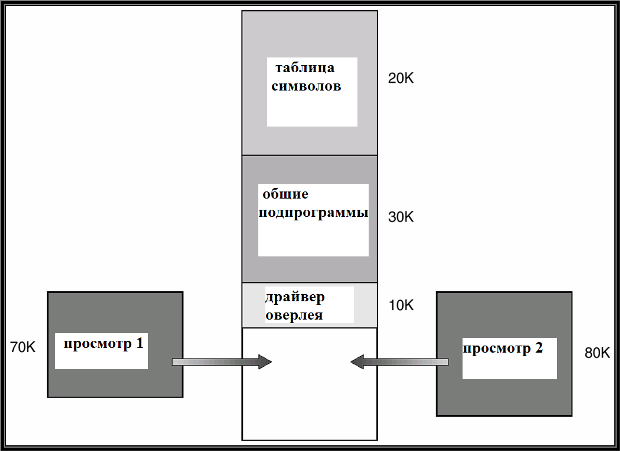
**Рис. 15.2.**Адресация с использованием регистра перемещения.

# Оверлейная структура двухпросмотрового ассемблера

### Оверлейная структура программы

Как мы уже отмечали во вводных лекциях, в ранних ОС, в особенности – для персональных компьютеров, для *пользовательского процесса* были вынужденно введены очень жесткие ограничения по памяти, - например, в MS *DOS* – не более 640 *килобайт*. При таком дефиците основной памяти, если *программа* оказывается настолько велика, что полностью не помещается в *память* максимально разрешенного объема, необходимо предпринимать специальные меры при разработке программы, чтобы разбить ее на непересекающиеся группы модулей, такие. что в каждой группе модули логически взаимосвязаны и должны присутствовать в памяти одновременно, модули же разных групп не обязательно должны вместе загружаться в *память*. Во время исполнения такой программы должен использоваться специальный системный механизм, называемый **оверлейная структура (** **overlay** ,дословно – **наложение** ), обеспечивающий поочередную загрузку в одну и ту же область памяти то одной, то другой исполняемой группы модулей. Простая *программа*, которая выполняет эти действия, называется **драйвер оверлея (** **overlay driver** **)**.Интегрированная *среда разработки* Турбо *Паскаль* обеспечивала специальные опции компилятора, которые позволяли явно указывать модули, входящие в каждый оверлей.

Типичный для ранних компьютеров и ОС пример программы с оверлейной структурой – двухпросмотровый *ассемблер*. На первом просмотре он преобразует исходный ассемблерный код в промежуточное *представление*, которое *программа* второго просмотра *ассемблера* получает на входе. Полностью весь *ассемблер* (оба просмотра) в *память* не помещался, и пришлось применить оверлейную структуру. Данный пример иллюстрируется на [рис. 15.3](https://intuit.ru/studies/courses/641/497/lecture/11294?page=2#image.15.3).



**Рис. 15.3.**Оверлейная структура двухпросмотрового ассемблера.

# откачка и подкачка. файл откачки. организовано смежное распределение памяти. Схема откачки и подкачки

### Введение

*Страничная организация* памяти (*paging*) – наиболее распространенная стратегия *управления памятью*, используемая практически во всех операционных системах. В данной лекции рассматриваются общие проблемы *управления памятью*, принципы *страничной организации* и ее различные формы.

### Откачка и подкачка

*Пользовательский процесс* может находиться в различных состояниях во время обработки системой. В частности, процесс может быть некоторое время неактивным, если, например, он исполняется в режиме разделения времени, и *пользователь* за терминалом обдумывает следующую команду или редактирует исходный код своей программы. В подобных случаях процесс может быть откачан операционной системой на *диск*, в связи с тем, что занимаемая им *память* оказывается необходимой в данный момент для другого, *активного, процесса*.

**Откачка и подкачка (swapping) –**это действия операционной системы *по* **откачке (записи)**образа неактивного процесса на *диск* или **подкачке (считыванию)***активного процесса* в *основную память*. Необходимость выполнения подобных действий вызвана нехваткой основной памяти.

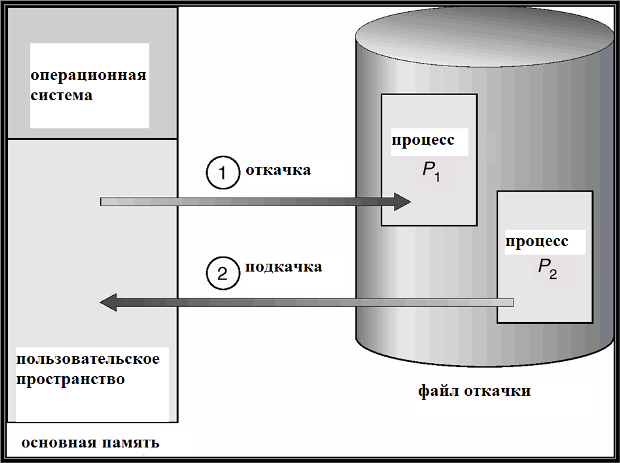
**Файл откачки (backing store) -**область дисковой памяти, используемая операционной системой для хранения образов откачанных процессов. *Файл* откачки организуется максимально эффективно: обеспечивается *прямой* *доступ* ко всем образам процессов в памяти (например, через таблицу *по* номеру процесса).

Популярная разновидность стратегии откачки и подкачки – **roll out / roll in**: откачка и подкачка на базе приоритетов; более *приоритетные процессы* исполняются, менее приоритетные – откачиваются на *диск*.

Наибольшие временные *затраты* на откачку – это *затраты* на передачу данных: полный образ процесса может занимать большую область памяти. Общее время откачки пропорционально размеру откачиваемых данных.

В распространенных ОС – *UNIX*, Linux, *Windows* и др. – реализованы различные стратегии откачки и подкачки.

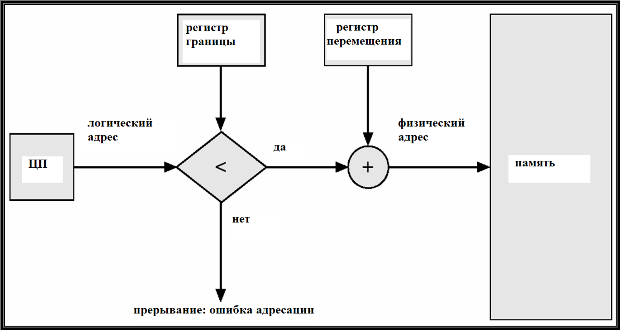
Схема откачки и подкачки изображена на [рис. 16.1](https://intuit.ru/studies/courses/641/497/lecture/11296?page=1#image.16.1).



**Рис. 16.1.**Схема откачки и подкачки.

### Смежное распределение памяти

Наиболее простая и распространенная стратегия *распределения памяти* – **смежное распределение памяти**– *распределение памяти* для *пользовательских процессов* в одной смежной области памяти. *Основная память* разбивается на две смежных части (*partitions*), которые "растут" навстречу друг другу: резидентная часть ОС и *вектор* прерываний – *по* меньшим адресам. Для *пользовательских процессов* *память* распределяется в одном и том же смежном участке памяти. Для каждого процесса *регистр* перемещения указывает на начало выделенной ему области памяти, *регистр* границы содержит длину диапазона *логических адресов*. Каждый *логический адрес* должен быть меньше содержимого регистра границы. *Физический адрес* вычисляется аппаратно как сумма *логического адреса* и значения регистра перемещения. Схема адресации с аппаратной поддержкой регистров перемещения и границы изображена на [рис. 16.2](https://intuit.ru/studies/courses/641/497/lecture/11296?page=1#image.16.2).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_2.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_2.png)  
**Рис. 16.2.**Адресация с аппаратной поддержкой регистров перемещения и границы.

### Общая задача распределения памяти и стратегии ее решения

В общем случае, в операционных системах может использоваться смежное *распределение памяти* в **нескольких**смежных областях. Свободная область – это смежный блок свободной памяти. Свободные области могут быть произвольно разбросаны *по* памяти. При загрузке процесса ему предоставляется *память* из любой свободной смежной области, которая достаточно велика для его размещения. При этом *операционная система* хранит **список свободных областей**памяти и **список занятых областей**памяти. Все эти области могут быть произвольно расположены в памяти и иметь различную длину.

Возникает **общая задача распределения памяти**: Имеется *список* свободных областей памяти и *список* занятых областей разного размера. Разработать и реализовать оптимальный (*по* некоторому критерию) *алгоритм* выделения свободного смежного участка памяти длины **n**(слов или байтов).

Для решения данной задачи применяются следующие стратегии: **метод первого подходящего (first-fit), метод наиболее подходящего (best-fit)**и **метод наименее подходящего (worst-fit)**.Рассмотрим каждую из них подробнее.

**Метод первого подходящего**:Выбирается первый *по* списку свободный участок подходящего размера (не меньшего, чем **n** ). На первый взгляд, данная стратегия оптимальна, но далее мы увидим, что это не всегда так.

**Метод наиболее подходящего**:Выбирается из списка наиболее подходящий свободный участок (минимального размера, не меньшего, чем **n** ). В отличие от предыдущего метода, требует просмотра всего списка, если *список* не упорядочен *по* размеру областей. Применение метода приводит к образованию оставшейся части самого маленького размера.

**Метод наименее подходящего**: Выбирается из списка подходящая область **наибольшего**размера. Почему наибольшего? Чтобы избежать фрагментации (проблема фрагментации подробно рассмотрена далее в данной лекции).

Применение первой и второй стратегий лучше со следующих точек зрения: скорость выполнения и минимальность объема использованной памяти. Однако их применение может создать фрагментацию.

### Фрагментация

**Фрагментация**– это дробление памяти на мелкие не смежные свободные области маленького размера. Фрагментация возникает после выполнения системой большого числа запросов на *память*, таких, что размеры подходящих свободных участков памяти оказываются немного больше, чем требуемые. Например, если имеется 100 смежных свободных областей памяти *по* 1000 слов, то после выполнения 100 запросов на *память* *по* 999 слов каждый в списке свободной памяти останутся 1000 областей *по* одному слову.

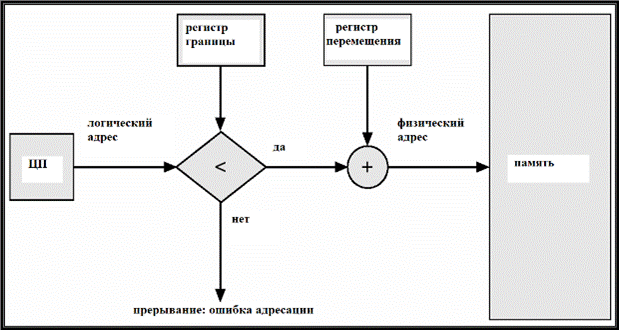
Фрагментация бывает **внутренняя**и **внешняя**. При **внешней фрагментации**имеется достаточно большая область свободной памяти, но она не является непрерывной. **Внутренняя фрагментация**может возникнуть вследствие применения системой специфической стратегии выделения памяти, при которой фактически в ответ на *запрос* *память* выделяется несколько большего размера, чем требуется, - например, с точностью до **страницы (листа** ), размер которого – степень двойки. *Страничная организация* памяти подробно рассматривается далее в данной лекции.

Внешняя фрагментация может быть уменьшена или ликвидирована путем применения **компактировки (compaction) –**сдвига или перемешивания памяти с целью объединения всех не смежных свободных областей в один непрерывный блок. Компактировка может выполняться либо простым сдвигом всех свободных областей памяти, либо путем перестановки занятых областей, с выбором на каждом шаге подходящей свободной области методом наиболее подходящего. Компактировка возможна, только если *связывание адресов* и перемещение (см. лекцию 15) происходит динамически. Компактировка выполняется во время исполнения программы.

При компактировке памяти и анализе свободных областей может быть выявлена **проблема зависшей задачи**: какая-либо задача может "застрять" в памяти, так как выполняет ввод-вывод в свою область памяти (*по* этой причине откачать ее невозможно). Решение данной проблемы: **ввод-вывод должен выполняться только в специальные буфера**, выделяемой для этой цели операционной системой.

# Схема адресации с аппаратной поддержкой регистров перемещения и границы

Наиболее простая и распространенная стратегия распределения памяти – смежное распределение памяти– распределение памяти для пользовательских процессов в одной смежной области памяти. Основная пмять разбивается на две смежных части (partitions), которые "растут" навстречу друг другу: резидентная часть ОС и вектор прерываний – по меньшим адресам, пользовательские процессы – по адресам. Для пользовательских процессов память распределяется в одном и том же смежном участке памяти. Для каждого процесса регистр перемещения указывает на начало выделенной ему области памяти, регистр границы содержит длину диапазона логических адресов. Каждый логический адрес должен быть меньше содержимого регистра границы. Физический адрес вычисляется аппаратно как сумма логического адреса и значения регистра перемещения. Схема адресации с аппаратной поддержкой регистров перемещения и границы изображена на рис. 16.2.



увеличить изображение

Рис. 16.2. Адресация с аппаратной поддержкой регистров перемещения и границы.

Общая задача распределения памяти и стратегии ее решения

В общем случае, в операционных системах может использоваться смежное распределение памяти в несколькихсмежных областях. Свободная область – это смежный блок свободной памяти. Свободные области могут быть произвольно разбросаны по памяти. При загрузке процесса ему предоставляется память из любой свободной смежной области, которая достаточно велика для его размещения. При этом операционная система хранит список свободных областейпамяти и список занятых областейпамяти. Все эти области могут быть произвольно расположены в памяти и иметь различную длину.

Возникает общая задача распределения памяти: Имеется список свободных областей памяти и список занятых областей разного размера. Разработать и реализовать оптимальный (по некоторому критерию) алгоритм выделения свободного смежного участка памяти длины n(слов или байтов).

Для решения данной задачи применяются следующие стратегии: метод первого подходящего (first-fit), метод наиболее подходящего (best-fit)и метод наименее подходящего (worst-fit).Рассмотрим каждую из них подробнее.

Метод первого подходящего:Выбирается первый по списку свободный участок подходящего размера (не меньшего, чем n). На первый взгляд, данная стратегия оптимальна, но далее мы увидим, что это не всегда так.

Метод наиболее подходящего:Выбирается из списка наиболее подходящий свободный участок (минимального размера, не меньшего, чем n). В отличие от предыдущего метода, требует просмотра всего списка, если список не упорядочен по размеру областей. Применение метода приводит к образованию оставшейся части самого маленького размера.

Метод наименее подходящего: Выбирается из списка подходящая область наибольшегоразмера. Почему наибольшего? Чтобы избежать фрагментации (проблема фрагментации подробно рассмотрена далее в данной лекции).

Применение первой и второй стратегий лучше со следующих точек зрения: скорость выполнения и минимальность объема использованной памяти. Однако их применение может создать фрагментацию.

# Архитектура трансляции адресов при страничной организации

### Страничная организация

**Страничная организация (paging) –**стратегия *управления памятью*, при которой:

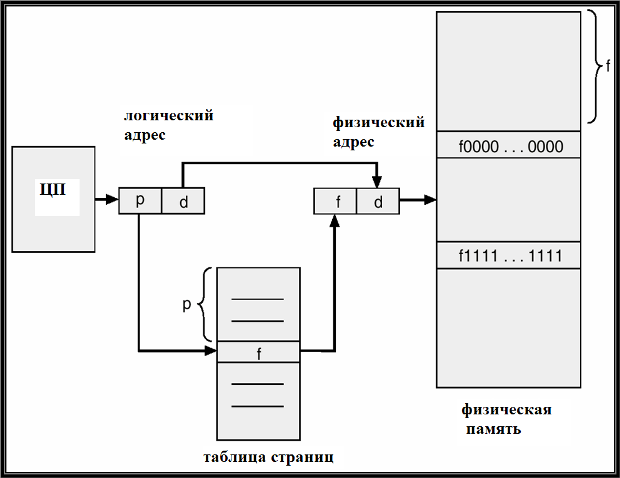
* *логическая память* делится на **страницы**– смежные области одинаковой длины, обычно – степень 2 (например, 512 слов);
* *физическая память*, соответственно, делится на **фреймы**такого же размера;
* распределение логической памяти происходит с точностью до страницы;
* *физическая память* процесса может не быть непрерывной;
* связь между логической и физической памятью процесса осуществляется с помощью **таблицы страниц**– системной структуры, выделяемой процессу для **трансляции**его **логических адресов в физические**.

При *страничной организации* ОС хранит информацию обо всех свободных фреймах. Поскольку *память* выделяется с точностью до страницы, возможна внутренняя фрагментация (см. п. 16.5).

Цели *страничной организации* – обеспечить возможность не смежного распределения физической памяти для процессов, а также расширить *пространство* логической памяти.

При *страничной организации* *логический адрес* обрабатывается системой особым образом – как структура **(p, d)**:его старшие разряды обозначают **номер страницы**, младшие – **смещение внутри страницы. Номер страницы (p)**трактуется как *индекс* в таблице страниц, соответствующий элемент которой содержит базовый **адрес начала страницы в физической памяти**. **Смещение внутри страницы (d)**добавляется к ее базовому адресу. В результате формируется *физический адрес*, передаваемый в *устройство управления памятью*.

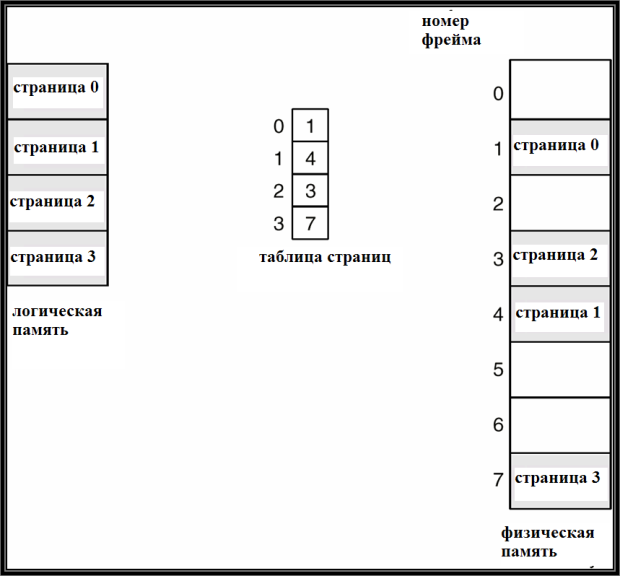
*Архитектура* *трансляции адресов* при *страничной организации* изображена на [рис. 16.3](https://intuit.ru/studies/courses/641/497/lecture/11296?page=2#image.16.3).



**Рис. 16.3.**Архитектура трансляции адресов при страничной организации.

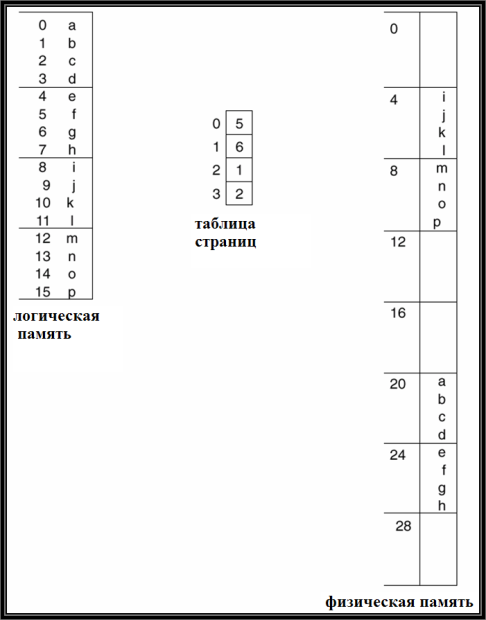
На [рис. 16.4](https://intuit.ru/studies/courses/641/497/lecture/11296?page=2#image.16.4) приведен пример *страничной организации*, который демонстрирует, что, в отличие от непрерывной логической памяти процесса, соответствующие фреймы страниц в основной памяти могут быть расположены не смежно: логической странице 0 соответствует *фрейм* 1, странице 1 – *фрейм* 4, странице 2 – *фрейм* 3, странице 3 – *фрейм* 7.

# таблицы страниц. Защита памяти



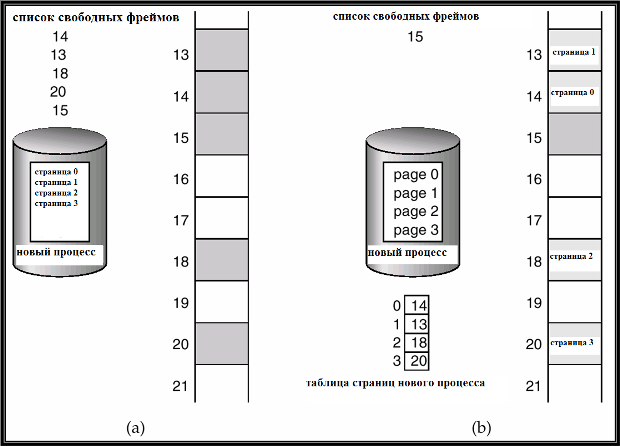
**Рис. 16.4.**Пример страничной организации.

На [рис. 16.5](https://intuit.ru/studies/courses/641/497/lecture/11296?page=2#image.16.5) приведен другой возможный пример *страничной организации*: логическая и *физическая память* разбита на блоки *по* 4 страницы подряд; в таблице страниц хранится не номер страницы, а номер блока страниц. Например, в элементе 0 *таблицы страниц* хранится номер блока 5, *по* которому *адрес* начала блока вычисляется домножением содержимого элемента *таблицы страниц* на размер блока, равный 4 (результат – 20).



**Рис. 16.5.**Пример страничной организации блоками по 4 страницы.

Использования списка свободных фреймов иллюстрируется на [рис. 16.6](https://intuit.ru/studies/courses/641/497/lecture/11296?page=2#image.16.6).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_6.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_6.png)  
**Рис. 16.6.**Список свободных фреймов.

Первоначально *список* состоит из 5 фреймов. При вводе в систему нового процесса с логической памятью из 4 страниц, после загрузки процесса в *память*, последовательные логические страницы процесса размещаются в первых *по* списку физических фреймах. В результате в списке свободных фреймов остается один элемент.

### Реализация таблицы страниц

**Использование ассоциативной памяти**.*Таблица страниц* – непрерывная область физической памяти. В системе имеется **базовый регистр таблицы страниц (page table base register – PTBR)**,указывающий на таблицу страниц и хранящий ее длину.

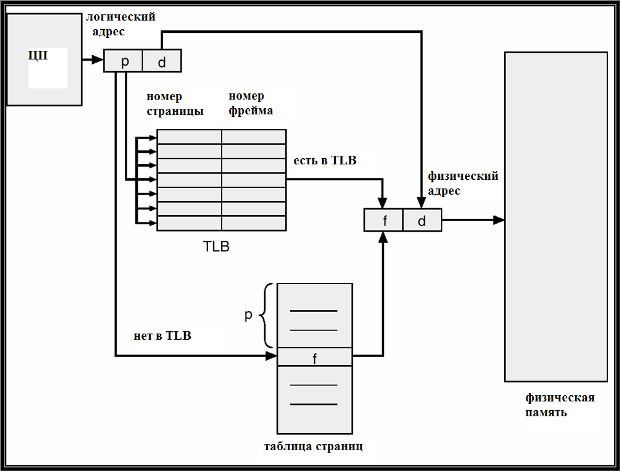
Таким образом, при *страничной организации* любой *доступ* к памяти требует фактически не одного, а двух обращений в *память* – одно в таблицу страниц, другое – непосредственно к данным или команде. В этом – некоторый недостаток и неэффективность *страничной организации*, *по* сравнению с более простыми *методами управления* памятью.

В системах с *теговой архитектурой*, например, "Эльбрус", *регистр* *таблицы страниц* ( **регистр таблицы страниц пользователя – РТСП)**содержит **дескриптор***таблицы страниц*, который, кроме ее адреса, содержит также ее длину.

Проблема двух обращений решается введением **ассоциативной памяти (cache) страниц**,называемой также **буфер трансляции адресов (translation lookaside buffer – TLB)**.*Ассоциативная память*, *по* существу, является ассоциативным списком пар вида: **(номер страницы, номер фрейма)**.Ее *быстродействие* значительно выше, чем у основной памяти и у регистров.

Схема *трансляции адресов* с использованием ассоциативной памяти изменяется: если **номер страницы**из *логического адреса* найден в ассоциативной памяти, то из ее элемента извлекается соответствующий **номер фрейма**.Если же номер страницы отсутствует в ассоциативной памяти, он выбирается обычным образом из *таблицы страниц*, но заносится в ассоциативную *память*. Таким образом, в ассоциативной памяти накапливается *информация* о наиболее часто используемых страницах.

Модифицированная схема *трансляции адресов* с использованием *TLB* иллюстрируется [рис. 16.7](https://intuit.ru/studies/courses/641/497/lecture/11296?page=3#image.16.7).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_7.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_7.png)  
**Рис. 16.7.**Схема трансляции адресов с использованием ассоциативной памяти.

### Оценка среднего времени доступа к памяти с использованием TLB

Предположим, что ассоциативный *поиск* в среднем требует \varepsilon единиц времени. Цикл памяти примем за 1. Введем также еще один показатель – число \alpha от 0 до 1 ( **hit ratio** ), характеризующее, сколько раз (в среднем) номер страницы будет найден в *TLB*, - эмпирическую *вероятность* нахождения номера страницы в ассоциативной памяти.

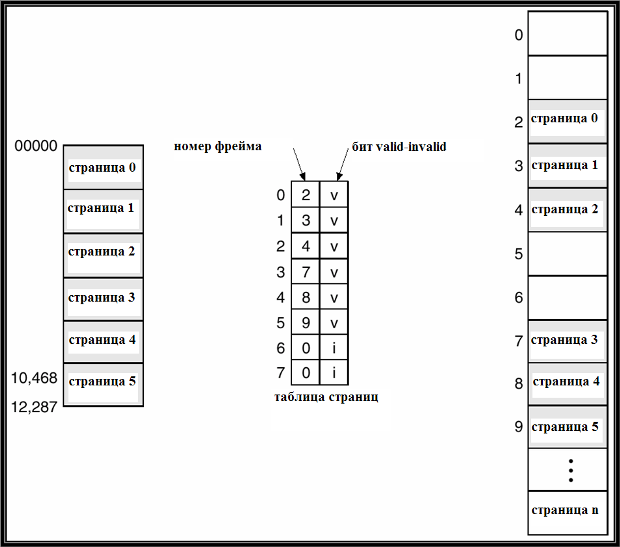
Вычислим *математическое ожидание* времени доступа – **Effective Access Time (EAT)**.*Вероятность* того, что номер страницы **не**будет найден в *TLB*, равна 1 – \alpha. Тогда получим:

EAT = (1 + \varepsilon ) \alpha  + (1 + \varepsilon  + 1) (1 – \alpha ) = 2 + \varepsilon  – \alpha

### Защита памяти

При адресации с помощью *страничной организации* возможно, что *логический адрес* сформирован неверно, и его номер страницы выходит за пределы логической памяти процесса. Защита от неверной адресации может быть реализована хранением и проверкой дополнительного **бита valid-invalid**в каждом элементе *таблицы страниц*. *Значение* **valid**указывает, что страница с данным номером принадлежит логической памяти процесса, *значение* **invalid**– что это не так.

Организация *защиты памяти* процесса с помощью бита *valid*-*invalid* иллюстрируется на [рис. 16.8](https://intuit.ru/studies/courses/641/497/lecture/11296?page=3#image.16.8).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_8.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_8.png)  
**Рис. 16.8.**Бит valid-invalid в таблице страниц.

В примере процесс имеет 6 логических страниц с номерами от 0 до 5. *Таблица страниц* имеет длину 8 (с элементами от 0 до 7). Элементы 6 и 7 не соответствуют логическим страницам процесса, поэтому в них биты *valid*-*invalid* установлены в *значение* **invalid**. Поэтому при попытке обращения *по* *логическому адресу* с номером страницы 6 или 7 произойдет *прерывание* *по* неверной адресации.

# Структура таблицы страниц. Хешированные таблицы страниц

### Структура таблицы страниц

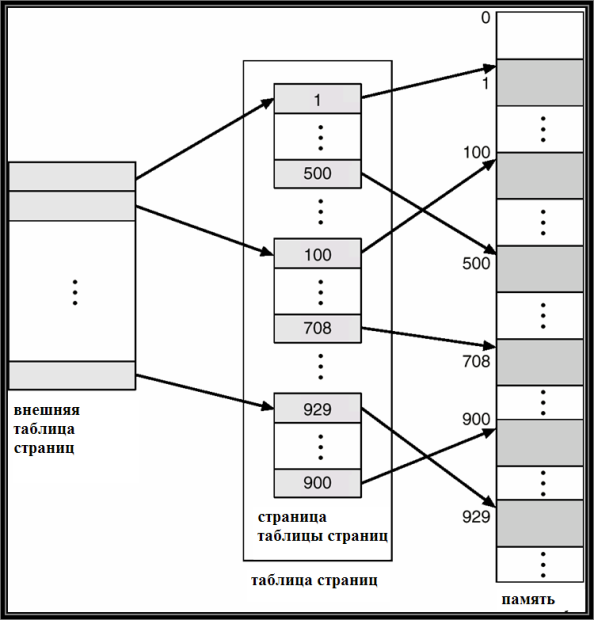
**Иерархические таблицы страниц**.*Таблицы страниц* в операционных системах могут быть *по*-разному организованы, при сохранении общих принципов их использования, описанных ранее. Рассмотрим далее три основных способа организации *таблиц страниц* – **иерархические таблицы страниц, хешированные таблицы страниц**и **инвертированные таблицы страниц**.

Чаще всего используются **иерархические таблицы страниц**.При их использовании логическое *адресное пространство* разбивается на несколько *таблиц страниц* (иначе говоря, используются **таблицы таблиц страниц** ). Наиболее простой и распространенный метод – **двухуровневая таблица страниц**.

При обычной организации *таблицы страниц*, *логический адрес* (для 32-разрядной архитектуры, при размере страницы 4 *килобайта* = 4096 байтов) разбивается на номер страницы (20 битов) и смещение внутри страницы (12 битов).

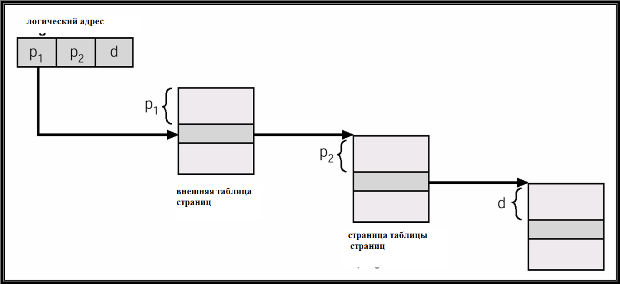
При двухуровневой организации *таблицы страниц*, *таблица страниц* верхнего уровня сама делится на страницы, поэтому *логический адрес* будет иметь вид: **(p1, p2, d)**,где **p1**– *индекс* во внешней таблице страниц, **p2**– смещение внутри страницы для внешней *таблицы страниц*, **d –**смещение внутри страницы (адресуемой *по* внутренней таблице страниц). При тех же предположениях об архитектуре и размере страницы, **p1**и **p2**будут занимать *по* 10 битов.

Организация двухуровневых *таблиц страниц* изображена на [рис. 16.9](https://intuit.ru/studies/courses/641/497/lecture/11296?page=3#image.16.9).



**Рис. 16.9.**Организация двухуровневых таблиц страниц.

Схема адресной трансляции *по* двухуровневой таблице страниц иллюстрируется [рис. 16.10](https://intuit.ru/studies/courses/641/497/lecture/11296?page=3#image.16.10).

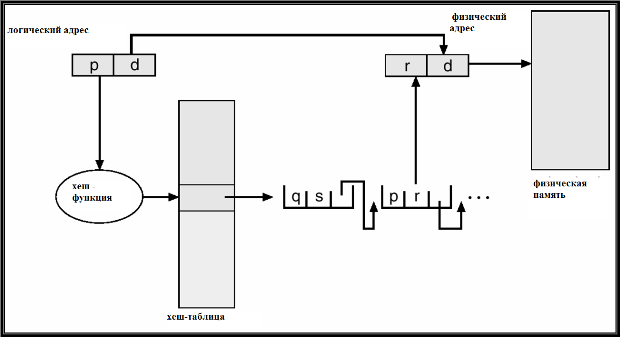
[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_10.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_10.png)  
**Рис. 16.10.**Схема адресной трансляции по двухуровневой таблице страниц.

### Хешированные таблицы страниц

Структура *таблицы страниц*, описанная в данном разделе, базируется на понятии **хеш-функции (hash function)**- целочисленной функции hash, определенной на элементах s некоторого пространства поиска S (строках, больших числах и др.) и принимающей значения из отрезка [0, H-1] где H – *натуральное число*. Общая идея поиска с помощью хеш-функций заключается в следующем: *пространство* поиска разбивается на H непересекающихся подмножеств (списков) Sh, в каждом из которых хранятся элементы, имеющие **одинаковое***значение* *хеш-функции*, равное h. Таким образом, применение *хеш-функции* позволяет сократить *поиск* в среднем в H раз: при поиске элемента s сначала вычисляем hash(s),а затем выполняем *поиск* только в списке Shash(s) Для облегчения доступа к хеш-спискам хранится также **хеш-оглавление**– *таблица*, индексируемая значением *хеш-функции*, каждый элемент которой содержит ссылку на начало соответствующего списка.

*Хешированные* *таблицы страниц* используются, если *адресное пространство* 64-битное или большей разрядности. Очевидно, что в этом случае, при прямолинейном подходе, размеры *таблиц страниц* оказались бы слишком велики. Поэтому используется другой метод. *Логический* номер страницы хешируется (т.е. для него вычисляется хеш-*функция*). Полученное число (реальный номер страницы) используется как *индекс* в хеш-оглавлении, ссылающемся на *список* страниц, хешируемых в один и тот же номер. В найденном таким образом списке и выполняется *поиск* нужной страницы. Данный способ организации *таблицы страниц* и схема адресации иллюстрируются [рис. 16.11](https://intuit.ru/studies/courses/641/497/lecture/11296?page=4#image.16.11).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_11.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/16/files/16_11.png)  
**Рис. 16.11.**Хешированные таблицы страниц.

# откачка и подкачка. файл откачки. Разделяемые страницы

### Откачка и подкачка

*Пользовательский процесс* может находиться в различных состояниях во время обработки системой. В частности, процесс может быть некоторое время неактивным, если, например, он исполняется в режиме разделения времени, и *пользователь* за терминалом обдумывает следующую команду или редактирует исходный код своей программы. В подобных случаях процесс может быть откачан операционной системой на *диск*, в связи с тем, что занимаемая им *память* оказывается необходимой в данный момент для другого, *активного, процесса*.

**Откачка и подкачка (swapping) –**это действия операционной системы *по* **откачке (записи)**образа неактивного процесса на *диск* или **подкачке (считыванию)***активного процесса* в *основную память*. Необходимость выполнения подобных действий вызвана нехваткой основной памяти.

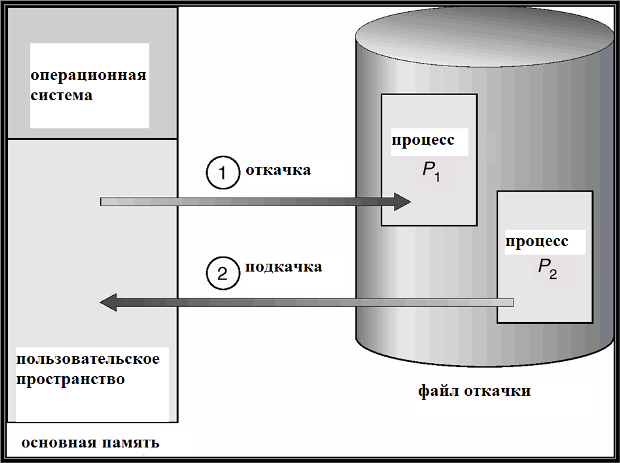
**Файл откачки (backing store) -**область дисковой памяти, используемая операционной системой для хранения образов откачанных процессов. *Файл* откачки организуется максимально эффективно: обеспечивается *прямой* *доступ* ко всем образам процессов в памяти (например, через таблицу *по* номеру процесса).

Популярная разновидность стратегии откачки и подкачки – **roll out / roll in**: откачка и подкачка на базе приоритетов; более *приоритетные процессы* исполняются, менее приоритетные – откачиваются на *диск*.

Наибольшие временные *затраты* на откачку – это *затраты* на передачу данных: полный образ процесса может занимать большую область памяти. Общее время откачки пропорционально размеру откачиваемых данных.

В распространенных ОС – *UNIX*, Linux, *Windows* и др. – реализованы различные стратегии откачки и подкачки.

Схема откачки и подкачки изображена на [рис. 16.1](https://intuit.ru/studies/courses/641/497/lecture/11296?page=1#image.16.1).



Файл подкачки (pagefile.sys) — это скрытый файл на жёстком диске компьютера, который используется операционной системой Windows по принципу оперативной памяти. Файл подкачки и оперативная память вместе составляют виртуальную память. По умолчанию операционная система Windows хранит файл подкачки в загрузочном разделе (раздел, в котором установлены операционная система и вспомогательные файлы). Обычно объём файла подкачки задаётся в 1,5 раза больше общего объёма оперативной памяти.

Файл подкачки необходим для того, чтобы разгрузить оперативную память в периоды её высокой загрузки. Если одновременно запустить несколько приложений, которые сильно загружают оперативную память, то часть этих приложений будет неактивна (они будут свёрнуты или попросту не будут использоваться в данный момент) и, как следствие, их данные будут выгружаться в менее быструю область, т. е. в файл подкачки, а данные активных в данный момент приложений будут находиться непосредственно в оперативной памяти. Когда производится обращение к неактивному приложению, данные из файла подкачки будут перемещены непосредственно в оперативную память, чтобы ускорить его работу. При острой нехватке оперативной памяти в системе файл подкачки используется напрямую, т. е. в него выгружаются и данные активных приложений. В случае если общий размер виртуальной памяти слишком мал, некоторые приложения могут работать нестабильно, а операционная система может выдать сообщение о том, что виртуальной памяти недостаточно. В таких ситуациях размер файла подкачки необходимо увеличить.

Если в вашем компьютере установлено более одного жёсткого диска, для улучшения быстродействия системы файл подкачки следует переместить в другой раздел на другом жёстком диске. В этом случае операционная система будет быстрее обрабатывать несколько запросов ввода-вывода.

Если в вашем компьютере установлен один жёсткий диск, также рекомендуется переместить файл подкачки в другой раздел жёсткого диска. Когда файл подкачки расположен в загрузочном разделе, операционная система Windows направляет запросы на чтение с диска и запись на диск как к системной папке, так и к файлу подкачки. После перемещения файла подкачки в другой раздел конкуренция между запросами записи и чтения уменьшается.

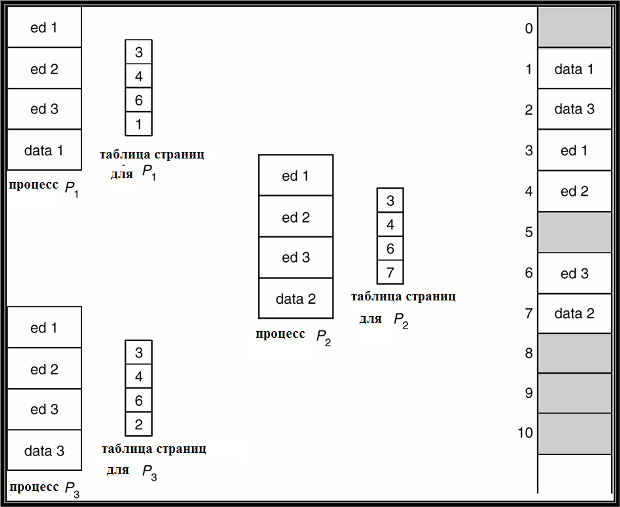
Если в Вашем компьютере установлены как HDD-, так и SSD-накопители, то файл подкачки рекомендуется расположить на одном из разделов HDD-накопителя. Это позволит уменьшить количество запросов на выполнение операций чтения/записи, что в свою очередь может благоприятно сказаться на отказоустойчивости SSD-накопителя.

Чтобы избежать фрагментации файла подкачки, рекомендуется создать на жёстком диске отдельный раздел, в котором будет располагаться только файл подкачки. Использование отдельного раздела для файла подкачки имеет следующее преимущество: в этом случае файл не разбивается на фрагменты. Файл подкачки, размещённый в разделе, где находятся другие данные, может быть разбит на фрагменты в процессе его увеличения для наращивания размеров виртуальной памяти. Дефрагментированный файл подкачки позволяет ускорить доступ к виртуальной памяти.

### Разделяемые страницы

**Разделяемые (совместно используемые) страницы**– логические страницы, используемые несколькими процессами и имеющие один и тот же номер в таблице страниц каждого из процессов. Данный механизм полезен для экономии памяти, так как позволяет загружать в *память* в единственном экземпляре не изменяемую информацию, необходимую нескольким процессам – например, код и массивы констант.

Пример использования разделяемых страниц тремя процессами приведен на [рис. 16.13](https://intuit.ru/studies/curriculums/4079/courses/497/lecture/11296?page=4#image.16.13). Три процесса используют одни и те же коды трех редакторов **ed1, ed2, ed3**, логические страницы которых имеют для обоих процессов номера 0, 1 и 2. Кроме того, каждый процесс использует свои индивидуальные данные **data1, data2, data3**.



**Рис. 16.13.**Разделяемые страницы.

# Принципы сегментной организации памяти. использования разделяемых сегментов

### Введение

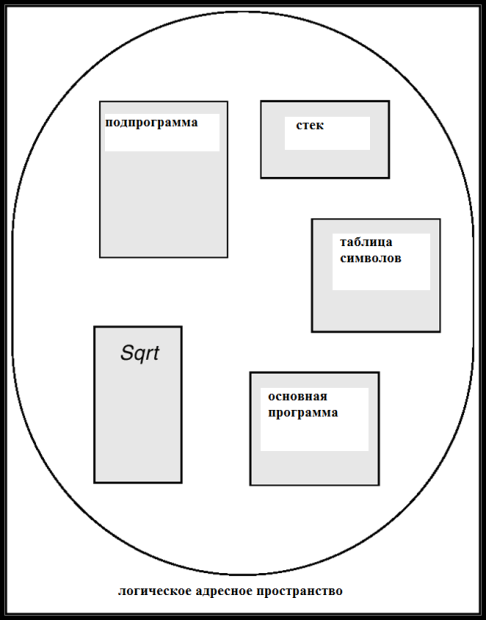
В лекции рассмотрена сегментная *организация памяти* – *альтернатива* *страничной организации*. Дано обоснование сегментной организации и ее связи с логической структурой программы. Рассмотрена смешанная – сегментно-страничная – *организация памяти*, применяемая во многих системах.

### Принципы сегментной организации памяти

Сегментная *организация памяти* (*segmentation*) **-**схема *распределения памяти* в виде сегментов переменной длины, соответствующая пользовательской трактовке *распределения памяти*, т.е. *логической структуре* программ и данных. С точки зрения пользователя (разработчика программы), *программа* – это набор **модулей**кода и данных, каждому из которых должен соответствовать свой **сегмент**в памяти. Сегмент – логическая *единица* *распределения памяти*, предназначенная для размещения в памяти одного модуля программного кода или данных. Например, в виде сегментов памяти могут быть представлены:

* *основная программа*;
* процедура;
* функция;
* метод;
* объект;
* набор локальных переменных;
* набор глобальных переменных;
* общий блок данных (например, COMMON-блок в языке *FORTRAN*);
* стек;
* *таблица символов*;
* массив.

[рис. 17.1](https://intuit.ru/studies/courses/641/497/lecture/11298?page=1#image.17.1) иллюстрирует данную точку зрения на программу как на набор сегментов в памяти.

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_1.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_1.png)  
**Рис. 17.1.**Программа как набор сегментов.

### [Архитектура сегментной организации памяти

Многие принципы архитектуры сегментной организации схожи с принципами *страничной организации* (см. ["Страничная организация памяти"](https://intuit.ru/studies/courses/641/497/lecture/11296)), однако во всех случаях приходится учитывать, что *длина* сегмента переменна, и хранить ее в явном виде в таблицах.

**Логический адрес при сегментной организации**памяти - пара:

<segment-number, offset>,

где *segment*-number – номер сегмента, offset – смещение в сегменте.

**Таблица сегментов**– служит для отображения логических адресов в физические при сегментной *организации памяти*. Каждый ее элемент содержит следующую информацию:

* **base –**начальный адрес сегмента в оперативной (физической) памяти;
* **limit**– длину сегмента.

**Базовый регистр таблицы сегментов - segment-table base register (STBR)**содержит *адрес* таблицы сегментов в памяти.

**Регистр длины таблицы сегментов - segment-table length register (STLR)**содержит число сегментов, используемое программой.

Номер сегмента s корректен, если s < STLR.

Перемещение (*relocation*) программ и данных при сегментной организации **динамическое**, т.е. выполняется во время исполнения программы с помощью таблицы сегментов.

Возможен общий *доступ* (sharing) нескольких процессов к одному и тому же сегменту, т.е. поддерживается концепция **разделяемых сегментов**. При этом *логический* номер общего сегмента для разных процессов будет одним и тем же.

**Стратегии распределения памяти**при сегментной организации: **метод первого подходящего**или **метод наиболее подходящего**(см. ["Страничная организация памяти"](https://intuit.ru/studies/courses/641/497/lecture/11296)). Метод наименее подходящего при сегментной организации смысла не имеет, так как он не улучшает ситуацию с фрагментацией (ввиду переменной длины сегментов). Соответственно, при сегментной организации возможна **внешняя фрагментация**, для борьбы с которой применяется компактировка (см. ["Страничная организация памяти"](https://intuit.ru/studies/courses/641/497/lecture/11296)).

**Защита**при сегментной организации организована аналогично защите при *страничной организации*, однако, ввиду того, что каждый сегмент выполняет определенную *логическую функцию* в программе, дополнительно с каждым сегментом связываются его **признаки защиты**.Таким образом, в каждом элементе таблицы сегментов хранятся:

* **validation-бит**(аналогично *страничной организации*): значение бита, равное 0, означает, что сегмент неверный, т.е. не принадлежит логической памяти процесса;
* **полномочия чтения (read) / записи (write) / исполнения (execute)**– каждое кодируется одним битом; значение бита, равное 0, означает, что процесс не имеет данных полномочий.

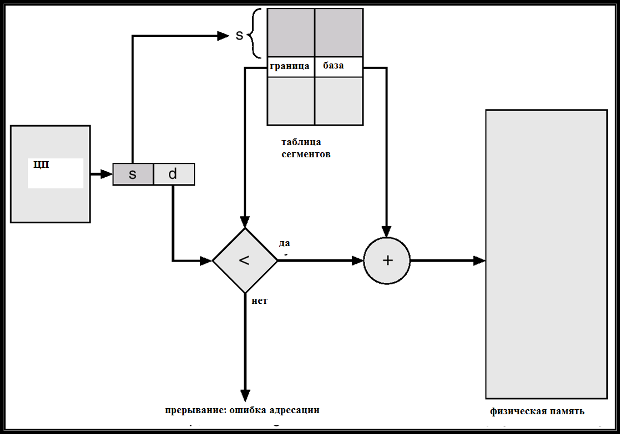
Например, если сегмент является сегментом данных, то система устанавливает в таблице сегментов *бит* защиты от исполнения равным 0. Если это сегмент кода, то целесообразна установка в 0 битов защиты от чтения и от записи.

Биты защиты связываются с сегментами. *Совместный доступ* к коду осуществляется на уровне сегментов.

В системе "Эльбрус" к стандартному набору признаков защиты был добавлен еще один: **защита от записи в сегмент адресной информации**(данный признак целесообразен, если, например, сегмент предназначен для записи в *файл*).

Поскольку *сегменты* различаются *по* длине, *распределение памяти* в виде сегментов – это общая задача *динамического распределения памяти* (см. ["Страничная организация памяти"](https://intuit.ru/studies/courses/641/497/lecture/11296)).

На [рис. 17.2](https://intuit.ru/studies/courses/641/497/lecture/11298?page=1#image.17.2) приведена схема адресации при сегментном распределении памяти. *Логический адрес* (s, d),где s – номер сегмента, d – смещение внутри сегмента, обрабатывается следующим образом. *По* номеру сегмента s происходит обращение в таблицу сегментов, и определяется база сегмента – его начальный *адрес* в основной памяти. Смещение d сравнивается с длиной (границей) сегмента. Если оно меньше, то оно складывается с базой, и в результате получается *физический адрес*, *по* которому и происходит обращение в *память*. Если смещение больше или равно базе, происходит *прерывание* – ошибка адресации. На схеме не показана еще одна проверка – для номера сегмента s проверяется, что он не превосходит значения регистра длины таблицы сегментов, иначе – *прерывание*.

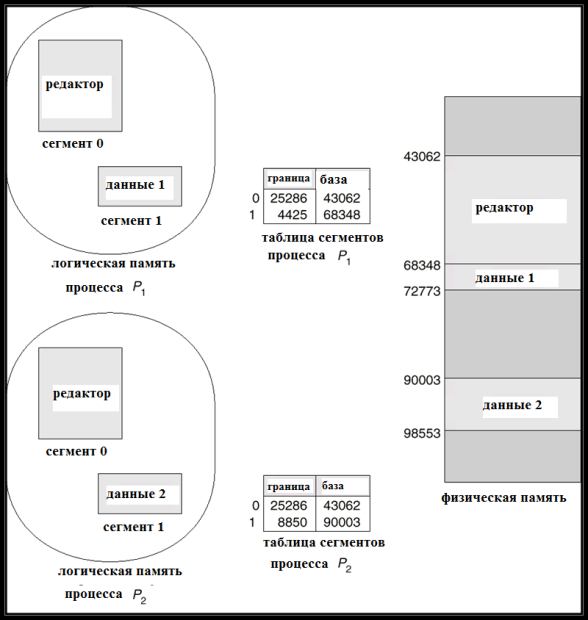
[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_2.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_2.png)  
**Рис. 17.2.**Схема адресации при сегментной организации памяти.

Пользователи систем *UNIX*, Linux, Solaris наверняка *по* ассоциации вспомнят часто встречающееся системное сообщение: **Segmentation violation (нарушение сегментации)**при прерывании. Причина подобного сообщения обычно следующая: *программа* пытается обратиться *по* **пустому (нулевому) указателю**, номер сегмента в котором, естественно, равен 0, а номер сегмента 0 считается в системе недопустимым. Таким образом, через проверку номера сегмента, *UNIX* защищает от такой часто встречающейся ошибки, как обращение *по* пустому указателю. Аналогично, *по* той же самой причине, если *программа* пытается практически любое другое число (например, 5) рассматривать как *адрес* и обращаться *по* нему, то также генерируется *прерывание* и сигнал **SIGSEGV (segmentation violation)**.

]

### Пример использования разделяемых сегментов

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_4.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_4.png)  
**Рис. 17.4.**Пример использования разделяемых сегментов.

В примере два процесса используют общий сегмент – код редактора, который имеет в таблицах сегментов обоих процессов один и тот же номер – 0. Кроме того, каждый из процессов имеет собственный сегмент данных с номером 1, однако эти *сегменты*, естественно, у обоих процессов разные.

# Схема адресации при сегментной организации памяти. Пример сегментной организации памяти

### Архитектура сегментной организации памяти

Многие принципы архитектуры сегментной организации схожи с принципами *страничной организации* (см. ["Страничная организация памяти"](https://intuit.ru/studies/courses/641/497/lecture/11296)), однако во всех случаях приходится учитывать, что *длина* сегмента переменна, и хранить ее в явном виде в таблицах.

**Логический адрес при сегментной организации**памяти - пара:

<segment-number, offset>,

где *segment*-number – номер сегмента, offset – смещение в сегменте.

**Таблица сегментов**– служит для отображения логических адресов в физические при сегментной *организации памяти*. Каждый ее элемент содержит следующую информацию:

* **base –**начальный адрес сегмента в оперативной (физической) памяти;
* **limit**– длину сегмента.

**Базовый регистр таблицы сегментов - segment-table base register (STBR)**содержит *адрес* таблицы сегментов в памяти.

**Регистр длины таблицы сегментов - segment-table length register (STLR)**содержит число сегментов, используемое программой.

Номер сегмента s корректен, если s < STLR.

Перемещение (*relocation*) программ и данных при сегментной организации **динамическое**, т.е. выполняется во время исполнения программы с помощью таблицы сегментов.

Возможен общий *доступ* (sharing) нескольких процессов к одному и тому же сегменту, т.е. поддерживается концепция **разделяемых сегментов**. При этом *логический* номер общего сегмента для разных процессов будет одним и тем же.

**Стратегии распределения памяти**при сегментной организации: **метод первого подходящего**или **метод наиболее подходящего**(см. ["Страничная организация памяти"](https://intuit.ru/studies/courses/641/497/lecture/11296)). Метод наименее подходящего при сегментной организации смысла не имеет, так как он не улучшает ситуацию с фрагментацией (ввиду переменной длины сегментов). Соответственно, при сегментной организации возможна **внешняя фрагментация**, для борьбы с которой применяется компактировка (см. ["Страничная организация памяти"](https://intuit.ru/studies/courses/641/497/lecture/11296)).

**Защита**при сегментной организации организована аналогично защите при *страничной организации*, однако, ввиду того, что каждый сегмент выполняет определенную *логическую функцию* в программе, дополнительно с каждым сегментом связываются его **признаки защиты**.Таким образом, в каждом элементе таблицы сегментов хранятся:

* **validation-бит**(аналогично *страничной организации*): значение бита, равное 0, означает, что сегмент неверный, т.е. не принадлежит логической памяти процесса;
* **полномочия чтения (read) / записи (write) / исполнения (execute)**– каждое кодируется одним битом; значение бита, равное 0, означает, что процесс не имеет данных полномочий.

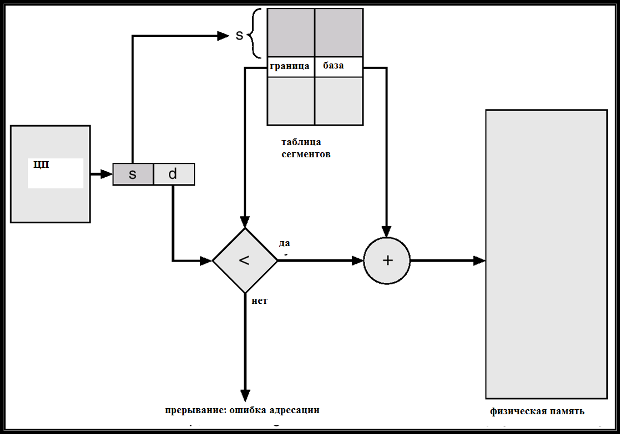
Например, если сегмент является сегментом данных, то система устанавливает в таблице сегментов *бит* защиты от исполнения равным 0. Если это сегмент кода, то целесообразна установка в 0 битов защиты от чтения и от записи.

Биты защиты связываются с сегментами. *Совместный доступ* к коду осуществляется на уровне сегментов.

В системе "Эльбрус" к стандартному набору признаков защиты был добавлен еще один: **защита от записи в сегмент адресной информации**(данный признак целесообразен, если, например, сегмент предназначен для записи в *файл*).

Поскольку *сегменты* различаются *по* длине, *распределение памяти* в виде сегментов – это общая задача *динамического распределения памяти* (см. ["Страничная организация памяти"](https://intuit.ru/studies/courses/641/497/lecture/11296)).

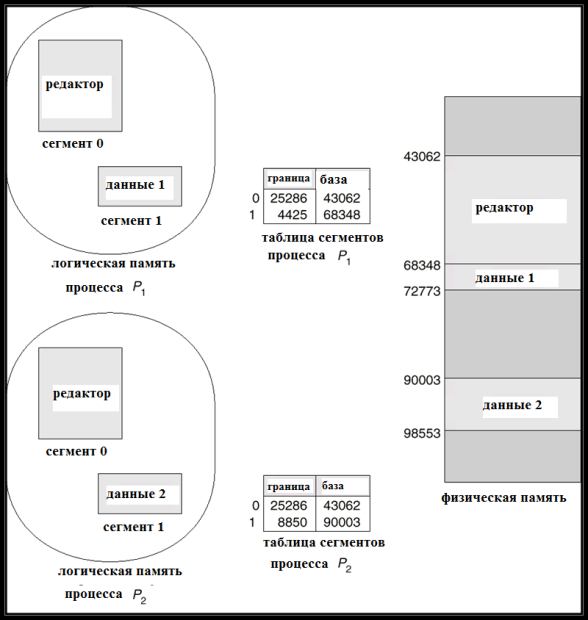
На [рис. 17.2](https://intuit.ru/studies/courses/641/497/lecture/11298?page=1#image.17.2) приведена схема адресации при сегментном распределении памяти. *Логический адрес* (s, d),где s – номер сегмента, d – смещение внутри сегмента, обрабатывается следующим образом. *По* номеру сегмента s происходит обращение в таблицу сегментов, и определяется база сегмента – его начальный *адрес* в основной памяти. Смещение d сравнивается с длиной (границей) сегмента. Если оно меньше, то оно складывается с базой, и в результате получается *физический адрес*, *по* которому и происходит обращение в *память*. Если смещение больше или равно базе, происходит *прерывание* – ошибка адресации. На схеме не показана еще одна проверка – для номера сегмента s проверяется, что он не превосходит значения регистра длины таблицы сегментов, иначе – *прерывание*.

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_2.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_2.png)  
**Рис. 17.2.**Схема адресации при сегментной организации памяти.

Пользователи систем *UNIX*, Linux, Solaris наверняка *по* ассоциации вспомнят часто встречающееся системное сообщение: **Segmentation violation (нарушение сегментации)**при прерывании. Причина подобного сообщения обычно следующая: *программа* пытается обратиться *по* **пустому (нулевому) указателю**, номер сегмента в котором, естественно, равен 0, а номер сегмента 0 считается в системе недопустимым. Таким образом, через проверку номера сегмента, *UNIX* защищает от такой часто встречающейся ошибки, как обращение *по* пустому указателю. Аналогично, *по* той же самой причине, если *программа* пытается практически любое другое число (например, 5) рассматривать как *адрес* и обращаться *по* нему, то также генерируется *прерывание* и сигнал **SIGSEGV (segmentation violation)**.

### Пример использования разделяемых сегментов

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_4.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/17/files/17_4.png)  
**Рис. 17.4.**Пример использования разделяемых сегментов.

В примере два процесса используют общий сегмент – код редактора, который имеет в таблицах сегментов обоих процессов один и тот же номер – 0. Кроме того, каждый из процессов имеет собственный сегмент данных с номером 1, однако эти *сегменты*, естественно, у обоих процессов разные.

# Понятие файла. Структура файла. Методы доступа к файлам

### **Введение**

История систем управления данными во *внешней памяти* начинается еще с магнитных лент, но современный облик они приобрели с появлением магнитных дисков. До этого каждая прикладная *программа* сама решала проблемы именования данных и их структуризации во *внешней памяти*. Это затрудняло поддержание на внешнем носителе нескольких архивов долговременно хранящейся информации. Историческим шагом стал переход к использованию централизованных систем управления *файлами*. Система управления *файлами* берет на себя распределение *внешней памяти*, *отображение* имен *файлов* в адреса *внешней памяти* и обеспечение доступа к данным.

**Файловая система** - это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во *внешней памяти*, и обеспечить пользователю удобный *интерфейс* при работе с такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском - прерогатива компонента системы ввода-вывода ОС, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. *Операции* записи или чтения *файла* концептуально проще, чем низкоуровневые *операции* работы с устройствами.

Основная идея использования *внешней памяти* состоит в следующем. ОС делит память на блоки фиксированного размера, например, 4096 байт. *Файл*, обычно представляющий собой неструктурированную последовательность однобайтовых записей, хранится в виде последовательности блоков (не обязательно смежных); каждый блок хранит целое число записей. В некоторых ОС (MS-DOS) адреса блоков, содержащих данные *файла*, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти. В других ОС (Unix) адреса блоков данных *файла* хранятся в отдельном блоке *внешней памяти* (так называемом индексе или индексном узле). Этот прием, называемый ***индексацией***, является наиболее распространенным для приложений, требующих произвольного доступа к записям *файлов* . *Индекс* *файла* состоит из списка элементов, каждый из которых содержит номер блока в *файле* и сведения о местоположении данного блока. Считывание очередного байта осуществляется с так называемой **текущей** позиции, которая характеризуется смещением от начала *файла*. Зная размер блока, легко вычислить номер блока, содержащего текущую позицию. *Адрес* же нужного блока диска можно затем извлечь из индекса *файла*. Базовой операцией, выполняемой по отношению к *файлу*, является чтение блока с диска и перенос его в *буфер*, находящийся в основной памяти.

*Файловая система* позволяет при помощи системы справочников ( *каталогов*, *директорий* ) связать уникальное имя *файла* с блоками вторичной памяти, содержащими данные *файла*. Иерархическая структура *каталогов*, используемая для управления *файлами*, может служить другим примером индексной структуры. В этом случае *каталоги* или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся *файловая система* компьютера представляет собой большой индексированный *файл*. Помимо собственно *файлов* и структур данных, используемых для управления *файлами* ( *каталоги*, дескрипторы *файлов*, различные таблицы распределения *внешней памяти* ), понятие "*файловая система*" включает *программные средства*, реализующие различные *операции над файлами*.

Перечислим **основные функции** файловой системы.

1. Идентификация *файлов*. Связывание имени *файла* с выделенным ему пространством *внешней памяти*.
2. Распределение *внешней памяти* между *файлами*. Для работы с конкретным *файлом* пользователю не требуется иметь информацию о местоположении этого *файла* на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ.
3. Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.
4. Обеспечение защиты от несанкционированного доступа.
5. Обеспечение совместного *доступа к файлам*, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.
6. Обеспечение высокой производительности.

Иногда говорят, что ***файл*** - это поименованный набор связанной информации, записанной во вторичную память. Для большинства пользователей *файловая система* - наиболее видимая часть ОС. Она предоставляет механизм для онлайнового хранения и доступа как к данным, так и к программам для всех пользователей системы. С точки зрения пользователя, ***файл*** - единица *внешней памяти*, то есть данные, записанные на диск, должны быть в составе какого-нибудь *файла* .

Важный аспект организации файловой системы - учет стоимости операций взаимодействия с вторичной памятью. Процесс считывания блока диска состоит из позиционирования считывающей головки над дорожкой, содержащей требуемый блок, ожидания, пока требуемый блок сделает оборот и окажется под головкой, и собственно считывания блока. Для этого требуется значительное время (десятки миллисекунд). В современных компьютерах обращение к диску осуществляется примерно в 100 000 раз медленнее, чем обращение к оперативной памяти. Таким образом, критерием вычислительной сложности алгоритмов, работающих с *внешней памятью*, является количество обращений к диску.

В данной лекции рассматриваются вопросы структуры, именования, *защиты файлов* ; *операции*, которые разрешается производить над *файлами* ; организация файлового архива (полного дерева справочников). Проблемы выделения дискового пространства, обеспечения производительной работы файловой системы и ряд других вопросов, интересующих разработчиков системы, вы найдете в следующей лекции.

### Общие сведения о файлах

#### Имена файлов

*Файлы* представляют собой абстрактные объекты. Их задача - хранить информацию, скрывая от пользователя детали работы с устройствами. Когда процесс создает *файл*, он дает ему имя. После завершения процесса *файл* продолжает существовать и через свое имя может быть доступен другим процессам.

Правила именования *файлов* зависят от ОС. Многие ОС поддерживают имена из **двух частей** (имя+расширение), например progr.c ( *файл*, содержащий текст программы на языке Си) или autoexec.bat ( *файл*, содержащий команды интерпретатора командного языка). Тип расширения *файла* позволяет ОС организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями. Обычно ОС накладывают некоторые ограничения, как на используемые в имени символы, так и на длину имени *файла*. В соответствии со стандартом POSIX, популярные ОС оперируют удобными для пользователя длинными именами (до 255 символов).

### **Структура файла**

В различных *системах приняты* различные точки зрения на структуру файлов. В ряде систем структура файла привязывалась к типу устройства, на котором он находится. В некоторых других системах структура файла была искусственно усложнена. Однако наиболее простую и унифицированную точку зрения из них предложили авторы системы *UNIX*: **файл – это последовательность слов или байтов**. Казалось бы, это очевидно, но преимущество данного подхода к файлам в том, что базовое *представление* файла и базовые *операции* над ним (*read*, *write*) не зависят от типа устройства. В свое время для программистов нашего поколения такой подход к файлам был откровением, после сложностей системы файлов *IBM* 360, а затем – "Эльбруса". Можно сказать, что файлы в своем развитии прошли *путь*, аналогичный развитию *архитектур компьютеров* – сначала в сторону значительных усложнений, затем – упрощения и *унификации*.

Файлы можно условно подразделять на файлы простой и сложной структуры (хотя точка зрения на структуру файла зависит от той программы, которая его обрабатывает).

Файлы **простой**структуры состоят из последовательности **записей (records) –**элементарных единиц, в терминах которых выполняются *операции* обмена с файлом. Записи могут быть:

* **строками**, если это **текстовый файл** ;
* двоичными данными **фиксированной длины** ;
* двоичными данными **переменной длины**.

Файлы **сложной структуры**могут быть самого разного вида, например:

* **отформатированным документом**Microsoft Office (такой файл, кроме собственно текста, содержит управляющие символы переключения шрифтов, цветов и т.д.);
* **загрузочным модулем**реального или виртуального двоичного кода, например, **portable executable (PE)**-файлом для платформы .NET; class-файлом для платформы Java; подобные файлы состоят из нескольких секций, содержат внутренние ссылки и таблицы и т.д.

Сложная структура файла может быть смоделирована записями путем добавления соответствующих *управляющих символов*.

Файлы интерпретируются операционной системой или программами их обработки.

### Методы доступа к файлам

Традиционно различаются файлы **последовательного доступа**и **прямого доступа**.*Файл* *последовательного доступа* – это *файл*, *доступ* к которому возможен только позиционированием на начало и конец и затем операциями обмена вида **считать**или **обновить следующую**( **предыдущую** ) **запись**. *Файл* прямого доступа – это *файл*, для которого возможен непосредственный *доступ* *по* номеру записи и операция обмена с явным указанием номера записи. В любом случае, при выполнении обмена с файлом всегда существует некоторая **текущая позиция***по* файлу, указывающая на некоторую *запись*, на позицию перед началом или после конца файла. В операциях над файлом *последовательного доступа* произвольная установка позиции не допускается, а разрешены только *операции*, автоматически передвигающие текущую позицию на следующую (предыдущую) *запись*.

Подобная особенность связана с различием устройств, на которых размещены файлы (например, магнитная лента – *по* сути дела, последовательное устройство), однако необходимость организации последовательных или прямых файлов может быть связана с сутью задачи.

*По*-видимому, *последовательный доступ* используется чаще: именно так происходит ввод данных, *вывод* результатов на *печать* или на экран.

Типичные *операции* *последовательного доступа*:

read next

write next

reset – установка на начало файла для чтения

*rewrite* – установка на начало файла для записи.

Типичные *операции* прямого доступа:

read n

write n

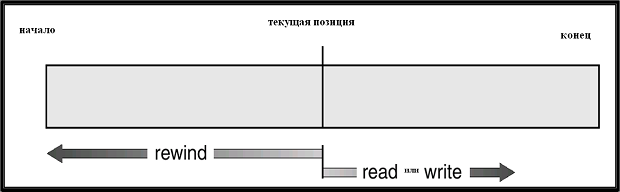
position to n – *позиционирование* на *запись* с номером n

read next

write next

*rewrite* n, где n - относительный номер блока (записи).

Описанная трактовка *последовательных файлов* и операций над ними иллюстрируется [рис. 19.1](https://intuit.ru/studies/courses/641/497/lecture/11302?page=2#image.19.1).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/19/files/19_2.png)

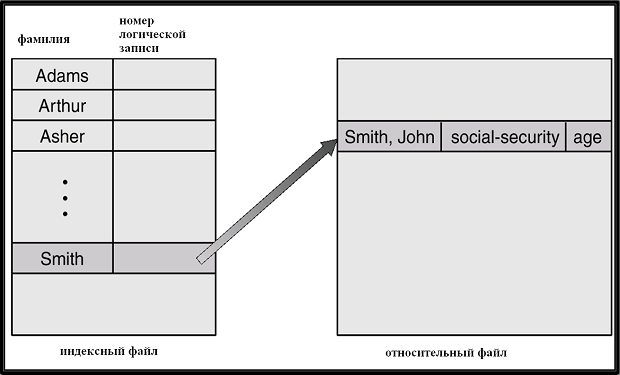
[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/19/files/19_2.png)  
**Рис. 19.1.**Файл последовательного доступа.

При необходимости можно смоделировать *операции* *последовательного доступа* для файла с прямым доступом. Способ моделирования показан в [таблица 19.2](https://intuit.ru/studies/courses/641/497/lecture/11302?page=2#table.19.2).

|  |  |
| --- | --- |
| Таблица 19.2. Моделирование *последовательного доступа* для файла с прямым доступом | |
| ***последовательный доступ*** | **реализация для файла с прямым доступом** |
| reset | *cp*=0; |
| read next | read *cp*; *cp*=*cp*+1; |
| write next | write *cp*; *cp*=*cp*+1; |

*Указатель* на текущую позицию, характерный для *последовательного доступа*, для файла прямого доступа моделируется переменной **cp**, а *операции* *последовательного доступа* – операциями прямого доступа с указанием **cp**в качестве номера блока.

Для ускорения доступа к файлу большого объема может использоваться **индексный файл**, содержащий ссылки на записи основного ( **относительного** ) файла. [рис. 19.2](https://intuit.ru/studies/courses/641/497/lecture/11302?page=2#image.19.2) изображен пример основного файла, содержащего упорядоченные *по* фамилиям анкетные данные людей. Индексный *файл* для данного основного файла содержит ссылки на первые логические записи анкет с заданной фамилией, например, Smith.

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/19/files/19_4.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/19/files/19_4.png)  
**Рис. 19.2.**Индексный файл и основной файл.

# Сетевые топологии. Стратегии маршрутизации

### Топология сетей

При создании систем и сетей передачи информации сетевые элементы объединяются на основе различных топологий. Выбор топологии зависит от типа сети:

* с коммутацией каналов или пакетов;
* транспортные или сети доступа;
* локальные или глобальные.

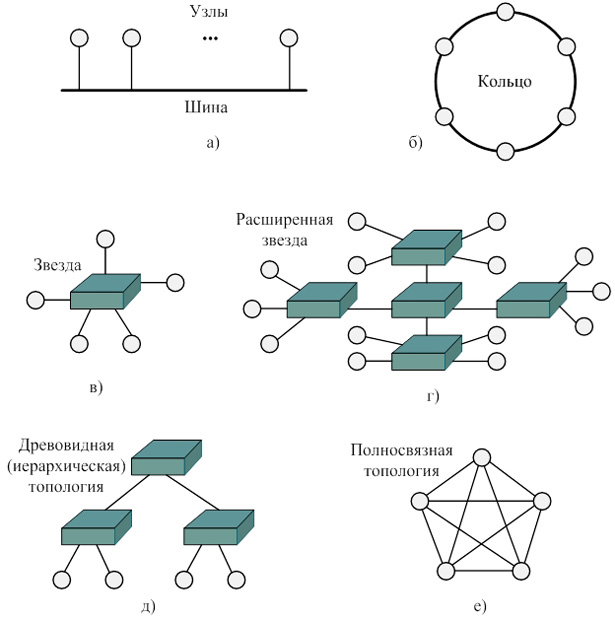
В различных типах сетей используются различные топологии и различные методы обеспечения надежности. Однако некоторые топологии используются практически во всех типах сетей. Далее рассмотрены широко распространенные *топологии локальных сетей*.

В инфокоммуникационных сетях различают физическую и логическую топологии сети. **Физическая топология** представляет собой наиболее общую структуру сети и отображает схему соединения сетевых элементов и узлов кабелями связи. **Логическая топология** показывает, как по сети передаются определенные единицы информации, и определяет *метод доступа* к сетевой среде передачи данных. В данном разделе рассматривается, главным образом, физическая *топология* локальных сетей.

В локальных сетях наибольшее распространение получили следующие физические топологии ([рис. 4.16](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.16)): *шина*, кольцо, *звезда*, расширенная *звезда*, древовидная (иерархическая) *топология*, а также полносвязная *топология*, где все узлы связаны между собой индивидуальными линиями связи.

Разделяемая (*shared*) линия или *среда передачи* данных, когда пользователи делят ресурсы линии связи между собой, снижает *стоимость* сети. Но в каждый момент времени линией может пользоваться только одна пара абонентов, из-за чего могут возникнуть очереди, а также коллизии.

**Топология шина** ([рис. 4.16](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.16)а) характеризуется тем, что передачу данных в данный момент времени может вести только один узел. Ожидание своей очереди на передачу данных является недостатком топологии. Если два узла одновременно начали передачу данных, то в сети возникает *коллизия*. При выходе какого-то узла из строя вся остальная *сеть* будет функционировать без изменений. Другими достоинствами топологии являются экономное расходование кабеля, простота, *надежность* и легкость расширения сети. *Топология* *шина* характерна для технологий ранних версий локальных сетей *Ethernet*, когда использовали коаксиальный *кабель*.

[](https://intuit.ru/EDI/25_07_20_1/1595629193-15985/tutorial/882/objects/4/files/04_16.jpg)

[увеличить изображение](https://intuit.ru/EDI/25_07_20_1/1595629193-15985/tutorial/882/objects/4/files/04_16.jpg)  
**Рис. 4.16.**Физические топологии локальных сетей

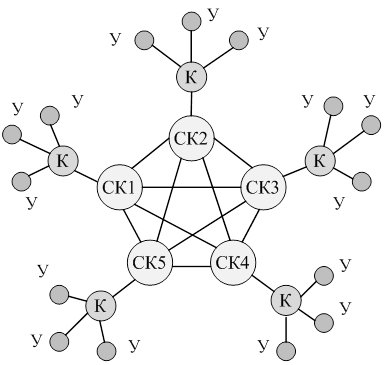
При использовании **топологии кольцо** ([рис. 4.16](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.16)б) сигналы передаются в одном направлении от узла к узлу. При выходе из стоя любого узла, прекращается функционирование всей сети, если не предусмотрен обход вышедшего из строя узла. Подобная физическая *топология* использовалась, например, в технологиях локальных сетей *Token Ring*, где для исключения коллизий реализован детерминированный *доступ* к разделяемой среде (кольцу). Передавать данные может только тот узел, который захватывает и удерживает специальный маркер, который циркулирует по кольцу. В настоящее время кольцевая *топология* широко используется в магистральных транспортных сетях.

**Топология звезда** ([рис. 4.16](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.16)в) требует применения центрального устройства, к которому подключены все узлы. *Выход* из стоя одного узла не влияет на работоспособность остальной сети. *Сеть* легко модифицируется путем подключения новых узлов, в ней легко организовать управление и обеспечить *безопасность*. Из недостатков можно отметить *уязвимость* центра и увеличенный расход кабеля по сравнению с топологией *шина*.

**Топология расширенная звезда** ([рис. 4.16](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.16)г) используется в современных крупных локальных сетях и сетях доступа, где широко распространены технологии GigabitEthernet. В качестве центрального устройства обычно устанавливается *коммутатор*. Разновидностью топологии расширенная *звезда* является **древовидная** или **иерархическая** ([рис. 4.16](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.16)д) *топология*, где функциональные возможности коммутаторов определяются уровнем иерархии.

Для повышения надежности и отказоустойчивости сетей их строят по **полносвязной топологии** ([рис. 4.16](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.16)е), где все узлы соединены между собой. Подобная *топология* характеризуется избыточностью, повышенным расходом кабеля, но все узлы постоянно связаны между собой, имеются запасные пути передачи данных.

На практике широко используется комбинация топологий. Например, *ядро* сети ([рис. 4.17](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.17)) содержит сетевые коммутаторы (СК1,…СК5), объединенные для повышения надежности и отказоустойчивости по полносвязной топологии. В целом *топология* сети представляет собой расширенную звезду или радиально-узловой способ построения сети, когда *конечные* узлы (У) подключены к концентраторам К, которые в свою *очередь*, соединены с сетевыми коммутаторами СК ядра сети. *Конечные* узлы (У) сети вместе с концентраторами (К) образуют *локальные сети*.



**Рис. 4.17.**Сеть передачи информации с комбинированной топологией

Совокупность локальных сетей образует **глобальную (составную, распределенную)** *сеть* (Wide *Area* *Network* - *WAN*). *Объединение* нескольких локальных сетей в **глобальную** *сеть* (Wide *Area* *Network* - *WAN*) происходит с помощью устройств и протоколов сетевого Уровня 3 семиуровневой эталонной модели *OSI* или уровня межсетевого взаимодействия четырехуровневой модели *TCP/IP*. Если *LAN* объединяют рабочие станции, периферию, терминалы и другое сетевое оборудование в одной аудитории или в одном здании, то *WAN* обеспечивают соединение *LAN* на широком географическом пространстве.

**Логическая топология** сети определяет, как узлы общаются через среду, т.е. как обеспечивается *управление доступом* к среде. Наиболее известные логические топологии: "*точка-точка*" (*point-to-point*), множественного доступа (multi access), широковещательная (*broadcast*) и маркерная (*token passing*).

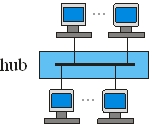
Логическая *топология* "*точка-точка*" обеспечивает передачу данных от одного узла до другого, независимо от промежуточных устройств между ними. Протокол управления передачей данных при такой топологии может быть очень простым, поскольку другие адресаты отсутствуют.

Логическая *топология* **множественного доступа** характерна для *Ethernet*-сетей, реализованных на многопортовых повторителях (*hub*). *Доступ* к разделяемой общей шине имеют все узлы, но в каждый момент времени передавать данные может только один узел. При этом остальные узлы могут только "слушать".

Использование **широковещательной** топологии определяет, что узел посылает свои данные всем другим узлам сетевой среды.

**Маркерная** логическая *топология*, также как *топология* множественного доступа реализует разделение общей среды. Однако, если в топологии multi-access *Ethernet*-сетей *доступ* к среде случайный (не детерминированный), то в маркерной топологии *доступ* к среде **детерминированный**. Электронный **маркер** (*token*) последовательно передается каждому узлу по кольцу. Узел, получивший маркер, может передавать данные в *сеть*. Если в узле нет данных для передачи, то он передает маркер следующему узлу и процесс повторяется. Топологию *token passing* используют сети: *Token Ring* и *Fiber* *Distributed* *Data* *Interface* (*FDDI*).

Широко известная *сетевая технология* *Ethernet* может использовать *концентраторы* (**hub**) и *кабель* "*витая пара*" ([рис. 4.18](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.18)). **Физическая** *топология* на [рис. 4.18](https://intuit.ru/studies/courses/3645/887/lecture/31131?page=5#image.4.18) представляет собой звезду, поскольку все компьютеры подключены к центральному устройству - концентратору (*hub*). **Логическая** же *топология* - **шина**, поскольку внутри концентратора все компьютеры подсоединены к общей магистрали. Поэтому выяснить, о какой топологии идет речь можно только из контекста.



**Рис. 4.18.**Топология: физическая - звезда, логическая – шина

### Стратегии маршрутизации

**Маршрутизация (routing) –***поиск* маршрута для каждого *сетевого пакета* и направление его *по* найденному маршруту. Для маршрутизации в сетях используется специальное сетевое оборудование – **маршрутизаторы (routers)**,которые обязательно должны использоваться в больших локальных сетях. Наиболее известные компании, выпускающие маршрутизаторы, - Cisco и *3COM*. Рассмотрим возможные стратегии маршрутизации.

**Фиксированная маршрутизация**. *Путь* от **A**к **B**задан заранее; он изменяется, только если им невозможно воспользоваться из-за отказов аппаратуры. При этом:

* Поскольку выбирается *кратчайший путь*, затраты на коммуникацию минимизированы.
* *Фиксированная маршрутизация* не может быть адаптирована к изменению загрузки.
* Обеспечивается получение сообщений в том же порядке, в каком они были посланы.

**Виртуальная цепочка**.*Путь* от **A**к **B**фиксируется на время одного сеанса. Различные сеансы, включающие сообщения от **A**в **B**, могут иметь различную маршрутизацию. Особенности данного способа маршрутизации:

* Это частичное средство адаптации к изменениям загрузки.
* Обеспечивается получение сообщений в том же порядке, в каком они были посланы.

**Динамическая маршрутизация**.*Путь* для *отправки сообщения* от A к B определяется только в момент *отправки данного* сообщения.

Обычно система посылает сообщение другой системе через соединение, наименее используемое в данный момент времени.

Метод *адаптирован* к изменениям загрузки, так как избегает отправки сообщений через интенсивно используемые соединения.

Сообщения при данном методе могут приходить в другом порядке. Эта проблема может быть решена путем присваивания *номера последовательности* каждому сообщению (что и реализовано в сетях *TCP/IP*).

# Уровни сетевых протоколов (ISO). Структура сетевого сообщения, согласно модели ISO. Уровни протокола TCP/IP.

### Введение

Самое основное в использовании сетей и сетевых технологий – сетевые протоколы, которым и посвящена данная лекция. В ней рассмотрены следующие вопросы:

* Протоколы коммуникации
* Устойчивость сетей
* Стратегии проектирования сетей.

### Протоколы коммуникации

**Модель ISO. Сетевой протокол**– это набор команд (операций) для взаимодействия узлов в сети. Ввиду сложности организации сетей, используемая модель сетевых протоколов содержит большое число уровней абстракции.

Согласно стандарту коммуникационной модели *ISO*, коммуникационная *сеть* подразделяется на следующие основные уровни (layers):

**Физический уровень (physical layer)**– механические и электрические устройства для передачи сигналов. Самый нижний уровень сетевой коммуникации. Включает сетевое оборудование - сетевые кабели, разъемы, *концентраторы* и т.д.

**Уровень (связывания) данных (Data link layer)**– обрабатывает фреймы ( **frames)**,или части пакетов фиксированной длины, включая обнаружение ошибок и восстановление после ошибок на физическом уровне.

**Сетевой уровень (network layer)**– обеспечивает соединение и маршрутизацию пакетов в *коммуникационной сети*, включая обработку адресов исходящих пакетов, *декодирование* адресов входящих пакетов и поддержку информации для маршрутизации для соответствующего ответа для изменения уровней загрузки.

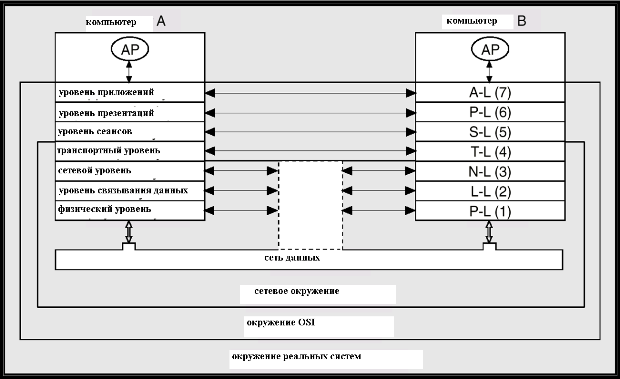
**Транспортный уровень (transport layer)**– отвечает за сетевой *доступ* нижнего уровня и за передачу сообщений между клиентами, включая разделение сообщений на пакеты, сопровождение порядка пакетов, *поток управления* и генерацию *физических адресов*.

**Уровень сеанса (session layer)**– реализует сеансы (sessions), или протоколы коммуникации между процессами.

**Уровень презентаций (presentation layer)**– инкапсулирует различие в форматах между различными системами в сети, включая преобразования символов и полудуплексную (*дуплексную) связь* (эхо-вывод).

**Уровень приложений (application layer)**– самый высокий *уровень модели* сетевых протоколов. Взаимодействует непосредственно с запросами на передачу файлов пользовательского уровня, протоколами удаленных входов и передачи электронной почты, а также со схемами *распределенных баз данных*.

Схема уровней сетевой коммуникации представлена на [рис. 23.1](https://intuit.ru/studies/courses/641/497/lecture/11310?page=1#image.23.1).

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/23/files/23_1.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/23/files/23_1.png)  
**Рис. 23.1.**Многоуровневая модель коммуникации ISO.

На [рис. 23.2](https://intuit.ru/studies/courses/641/497/lecture/11310?page=1#image.23.2) представлено содержание уровней сетевой коммуникации.



**Рис. 23.2.**Уровни сетевых протоколов (ISO).

На [рис. 23.3](https://intuit.ru/studies/courses/641/497/lecture/11310?page=1#image.23.3) изображена структура сетевого сообщения, согласно модели *ISO*.



**Рис. 23.3.**Структура сетевого сообщения, согласно модели ISO.

Сообщение имеет *заголовки* каждого уровня, начиная от уровня связывания данных, затем следует *тело сообщения* и признак конца сообщения (уровня связывания данных, отвечающего за *целостность* передачи сообщения). Таким образом, реализация каждого уровня абстракции обрабатывает сообщение, используя *заголовок сообщения* соответствующего уровня.

### Протокол TCP/IP

**IP-адреса. TCP/IP (Transmission Control Protocol / Internet Protocol)**– наиболее распространенное *семейство протоколов* сетевого и транспортного уровня, используемых в Интернете. Основоположники *TCP*/*IP* – Роберт Кан (Robert Kahn) и Винтон Серф (Vinton Cerf) -1972 – 1974.

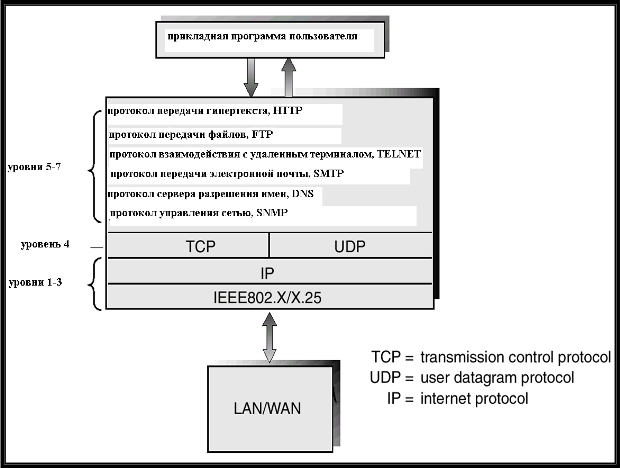
*Протокол TCP*/*IP* основан на использовании **IP-адресов**каждого **хоста**(компьютера), имеющих вид: **a.b.c.d**(все четыре числа – в диапазоне от 0 до 255) и обеспечивает пересылку по сети **пакетов (packets)**фиксированного размера, содержащих *адрес* получателя и номер пакета в сообщении. *TCP*/*IP* -протокол обеспечивает транспорт *сетевых пакетов*, *деление* сообщения на пакеты отправителем и сборку сообщения из пакетов получателем. *IP*-*адрес* может быть закреплен за компьютером постоянно *Интернет*-провайдером пользователя компьютера, либо присваивается компьютеру динамически (каждый раз – разный) при выходе в *Интернет*.

Более общее современное название *TCP*/*IP* - **Internet Protocol Suite**.Различаются более новая версия – *IPv6* и более старая – *IPv4*. В версии *IPv6* используется *IP*-*адрес* не из четырех, а из 6 чисел.

*Протокол TCP*/*IP* – *синхронный*, т.е. получатель ожидает получения каждого пакета и посылает отправителю подтверждение об этом. Другой вариант *Интернет*-протокола - **UDP/IP (Universal Datagram Protocol / Internet Protocol)**– асинхронный транспортный протокол, обеспечивающий обмен **датаграммами**– байтовыми массивами переменной длины; он считается менее надежным, чем *TCP*/*IP*, но работает быстрее, поэтому часто для быстрого обмена сообщениями в сетях предпочитают именно его.

Скорость *TCP*/*IP* не всегда удовлетворительна, ввиду огромного числа *IP*-узлов в Интернете. Для оптимизации связи между узлами сети применяются **Distributed Hash Tables (DHT) – распределенные хеш-таблицы**. В них собственная система имен узлов сети и более быстрого их поиска, чем с использованием *TCP*/*IP* протоколов, работающая "поверх" *TCP*/*IP*.

На [рис. 23.4](https://intuit.ru/studies/courses/641/497/lecture/11310?page=2#image.23.4) изображены уровни протокола *TCP*/*IP* и перечислены основные протоколы прикладного уровня, работающие поверх *TCP*/*IP* и UDP/*IP*, - *протокол передачи гипертекста* (основа функционирования World Wide *Web*), протоколы передачи файлов, передачи электронной почты, взаимодействия с удаленным терминалом, управления сетью.

[](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/23/files/23_4.png)

[увеличить изображение](https://intuit.ru/EDI/22_02_19_2/1550787571-32127/tutorial/901/objects/23/files/23_4.png)  
**Рис. 23.4.**Уровни протокола TCP/IP.