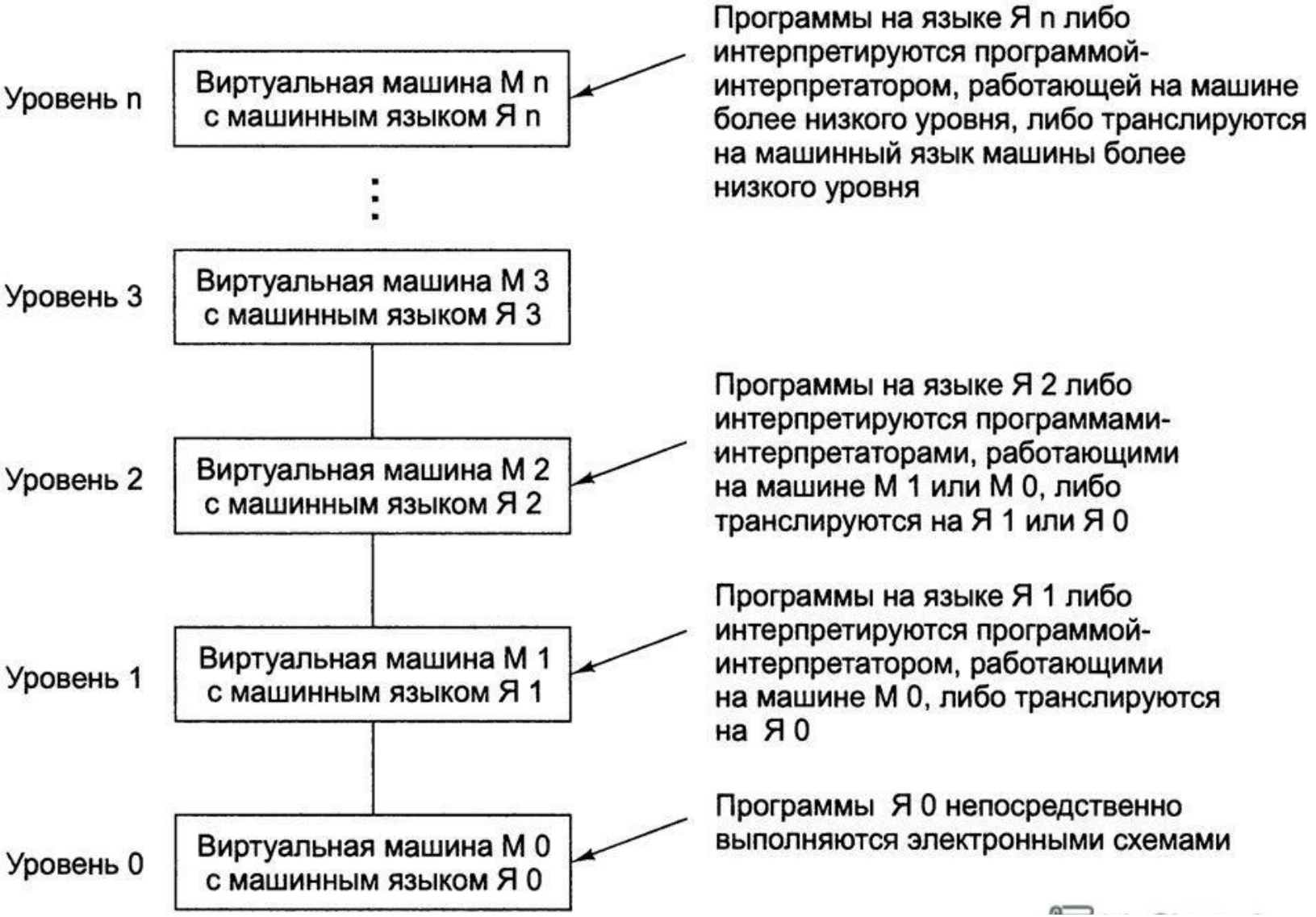
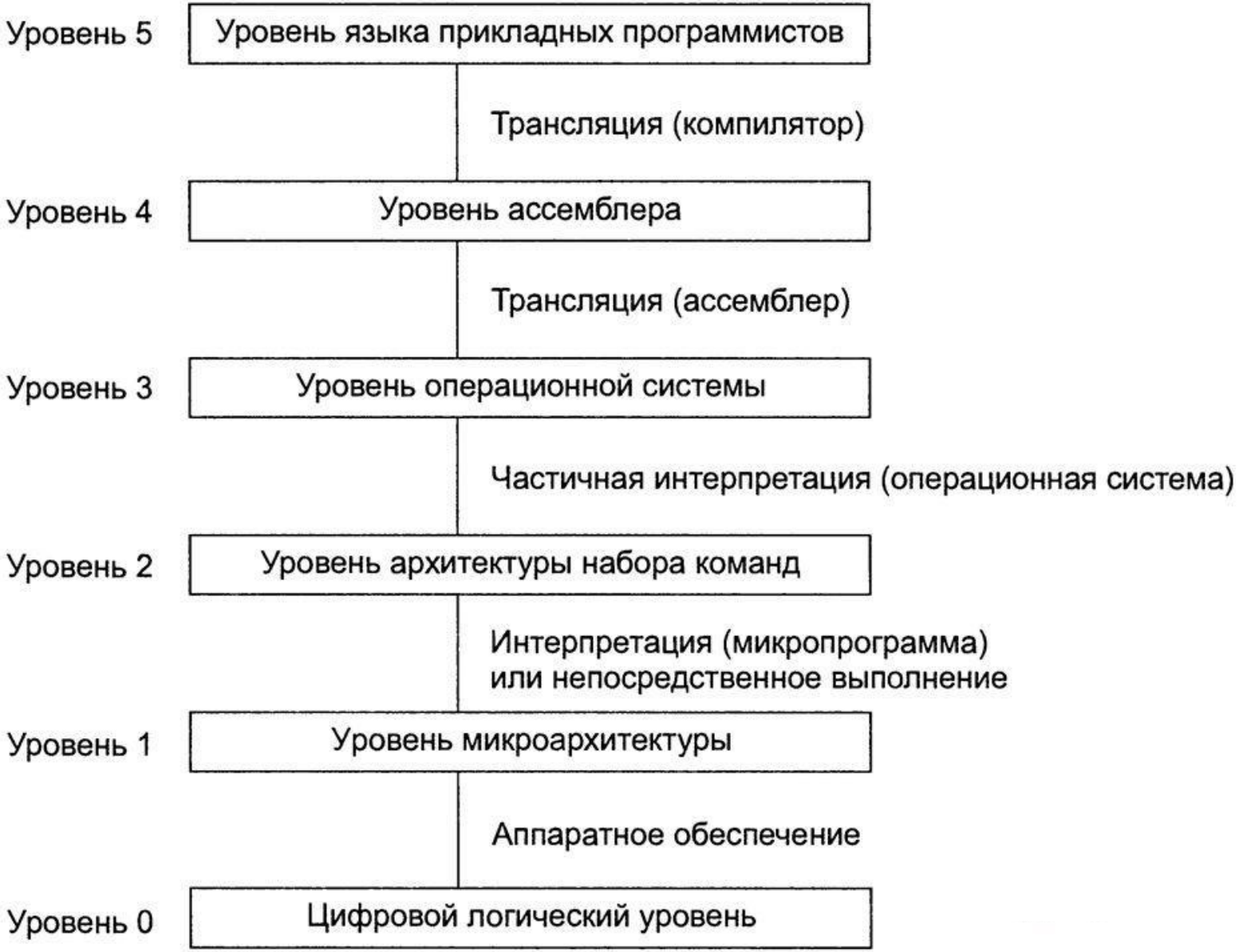
# Понятие операционной системы. Виртуальные машины.



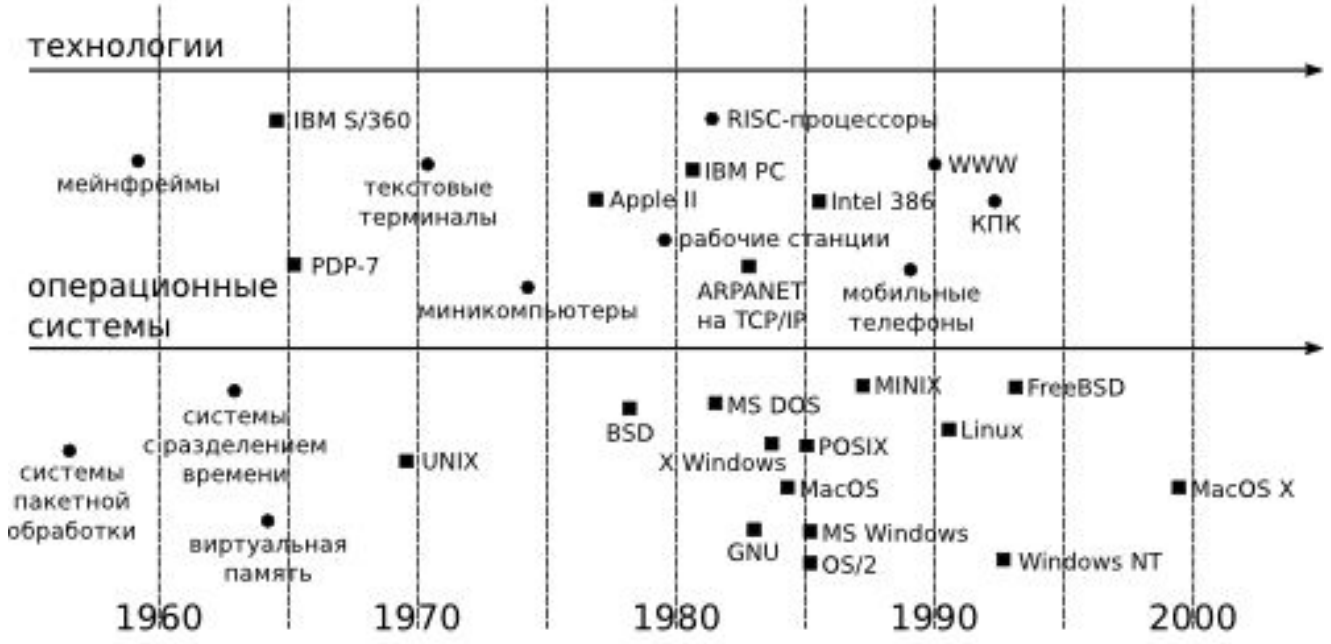

Операционная система была создана для того, чтобы автоматизировать работу оператора и скрыть от пользователя сложности общения с аппаратурой, предоставив ему более удобную систему команд.

Назначение ОС состоит в предоставлении пользователю/программисту некоторой расширенной виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальный компьютер или реальную сеть.




- Концепция сверху вниз рассматривающая операционную систему как удобный интерфейс пользователя.
- Концепция снизу вверх (альтернативный взгляд) дает представление об операционной системе как о механизме, присутствующем в компьютере для управления всеми компонентами этой сложнейшей системы. В рамках этой концепции - работа операционной системы заключается в обеспечении организованного и контролируемого распределения процессоров, памяти, дисков, принтеров, устройств ввода-вывода, датчиков времени и т.п. между различными программами, конкурирующими за право их использовать.

# История ОС




**1945-1955г.г.**

**Без операционных систем** (как правило, на них работала одна программа). Когда скорость выполнения программ и их количество стало увеличиваться, простои компьютера между запусками программ стали составлять значительное время.

**1955-1965г.г.**

**Системы пакетной обработки**, которые просто автоматизировали запуск одной программ за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем.

**1965-1980г.г.**

- Разработана многопользовательская система MULTICS
- мини-компьютеры (первый был выпущен в 1961г.), на которые была перенесена система MULTICS (в дальнейшем **UNIX**).

Так как появилось много разновидностей несовместимых UNIX(System V и BSD), был разработан стандарт **POSIX,** который определяет минимальный интерфейс системного вызова.

**1987г. - OS/2**

**1987г. - MINIX**

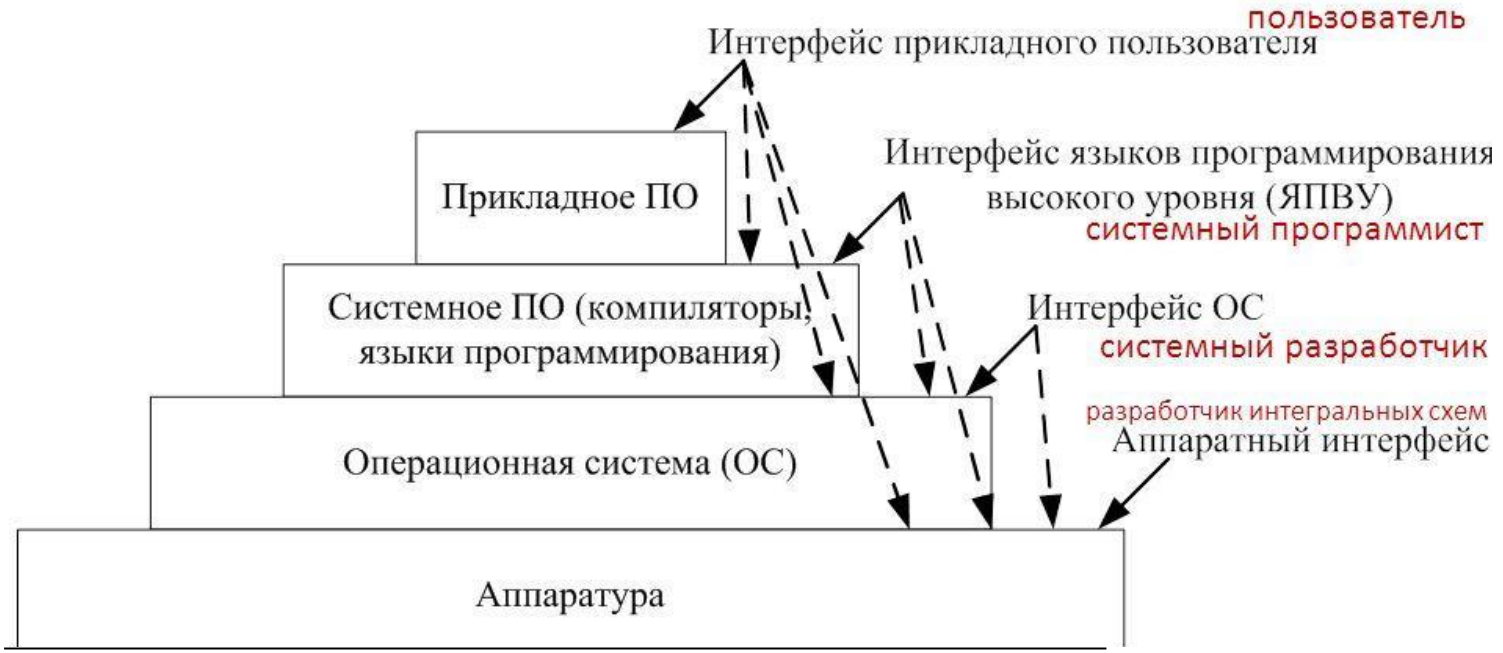
Была выпущена операционная система **MINIX**, она была построена на схеме **микро ядра**.

**1991г. – LINUX**

Была выпущена **LINUX**, в отличии от микроядерной MINIX она стала монолитной. Чуть позже вышла **FreeBSD** (основой для нее послужила BSD UNIX).

**1993г. - Windows NT**

# Иерархической структура программного и аппаратного обеспечения компьютера

# Назначение, состав и функции ОС

## ОС как виртуальная машина

ОС предоставляет пользователю **виртуальную машину**, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой реальной машины.

## ОС как система управления ресурсами:

Чтобы несколько программ могло работать с одним ресурсом (процессор, память), необходима **система управления ресурсами**.

Способы распределения ресурса:

- **Временной** - когда программы используют его по очереди (процессор (кванты)).
- **Пространственный** - программа получает часть ресурса (оперативна память и жесткий диск).

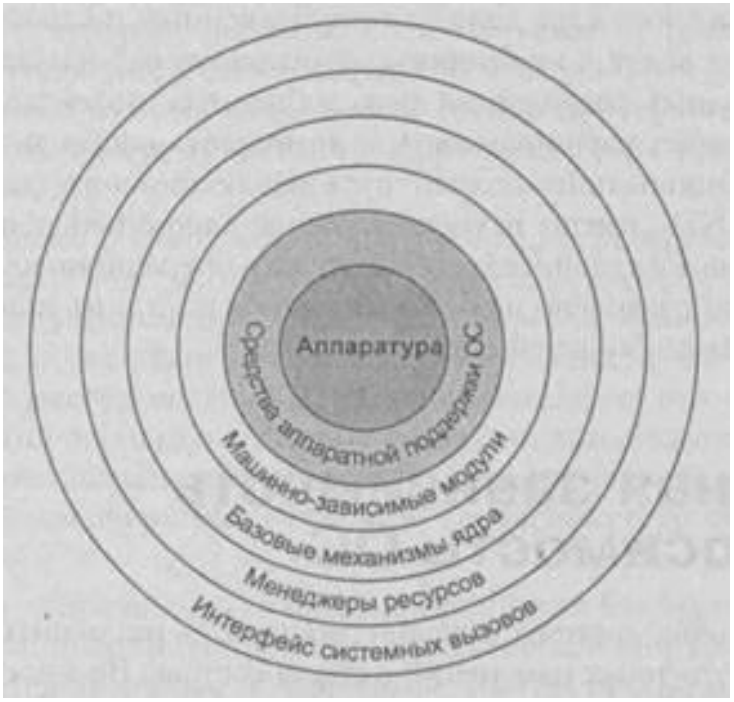
**Функции ОС**


Операционная системы
Подсистема управления ресурсами
Общие подсистемы
Управление процессами
Управление памятью
Управление файлами.
Управление внешними устройствами
Пользовательский интерфейс.
Интерфейс прикладного программирования.
Защита данных и администрирование

# Многоуровневая структура ОС

Вычислительную систему, работающую под управлением ОС, можно рассматривать как систему, состоящую из трех иерархически расположенных слоев: нижний слой образует аппаратура, промежуточный - ядро, а утилиты, обрабатывающие программы и приложения, составляют верхний слой системы.

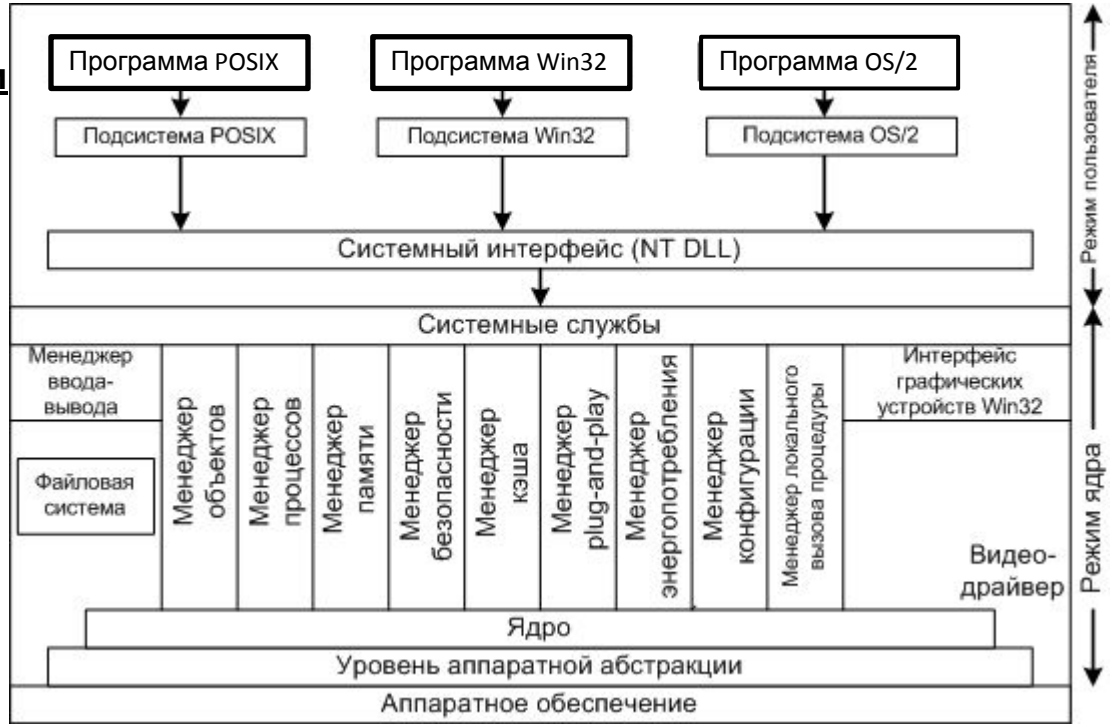
Многослойный подход - универсальный и эффективный способ декомпозиции сложных систем. В соответствии с этим подходом система состоит из иерархии слоев. Каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс



**Пример реализации многоуровневой модели UNIX.**


**Структура ОС UNIX**

**Пример реализации многоуровневой модели WINDOWS.**

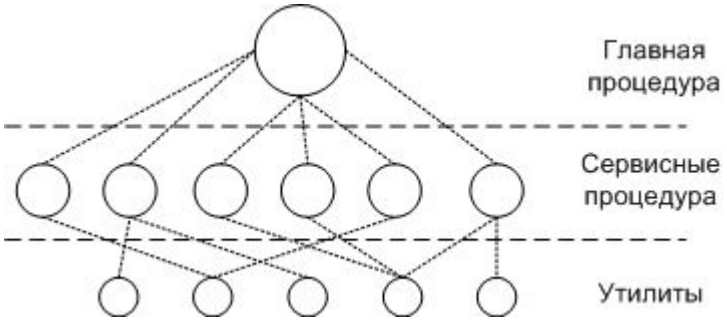



**Структура Windows 2000**

# Виды архитектуры ядер операционной системы

## Монолитное ядро


Все компоненты монолитного ядра находятся в одном адресном пространстве.



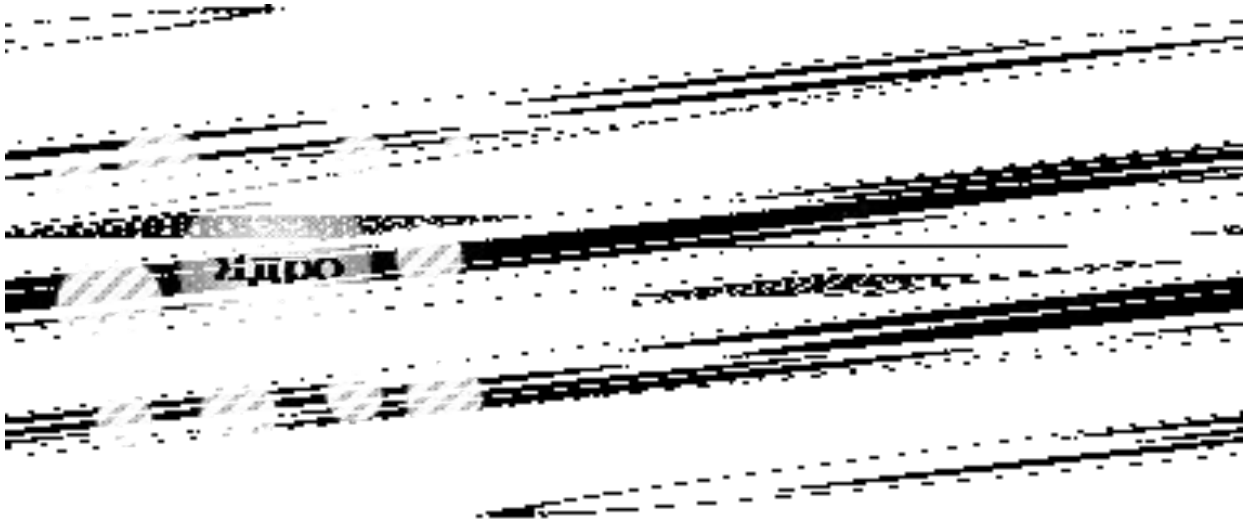

В этой модели для каждого системного вызова имеется одна сервисная процедура (например, читать из файла). Утилиты выполняют функции, которые нужны нескольким сервисным процедурам (например, для чтения и записи файла необходима утилита работы с диском).

Этапы обработки вызова:

- Принимается вызов
- Выполняется переход из режима пользователя в режим ядра
- ОС проверяет параметры вызова для того, чтобы определить, какой системный вызов должен быть выполнен
- После этого ОС обращается к таблице, содержащей ссылки на процедуры, и вызывает соответствующую процедуру

# Модульное ядро

Модификация монолитных ядер ОС, но в отличие от них модульное ядро не требует полной перекомпиляции ядра при изменения аппаратного обеспечения компьютера




Модули ОС, оформленные в виде утилит, системных обрабатывающих программ и библиотек, обычно загружаются в оперативную память только на время выполнения своих функций, то есть являются транзитными. Постоянно в оперативной памяти располагаются только самые необходимые коды ОС составляющие ее ядро. Такая организация ОС экономит оперативную память компьютера.

# Микроядро


Переход от Монолитного ядра к Микроядру

В развитии современных операционных систем наблюдается тенденция в сторону дальнейшего переноса задач из ядра в уровень пользовательских процессов, оставляя минимальное микроядро.

В этой модели вводятся два понятия:

1. **Серверный процесс** (который обрабатывает запросы)
2. **Клиентский процесс** (который посылает запросы)

В задачу ядра входит только управление связью между клиентами и серверами.

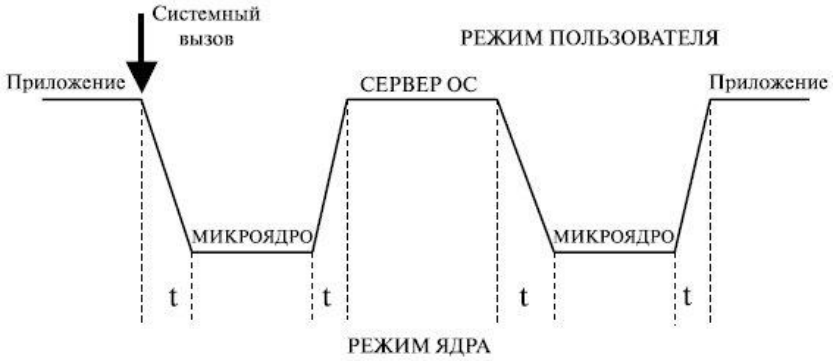
**Преимущества**:

- Малый код ядра и отдельных подсистем, и как следствие меньшее содержание ошибок.
- Ядро лучше защищено от вспомогательных процессов.
- Легко адаптируется к использованию в распределенной системе.

**Недостатки**:

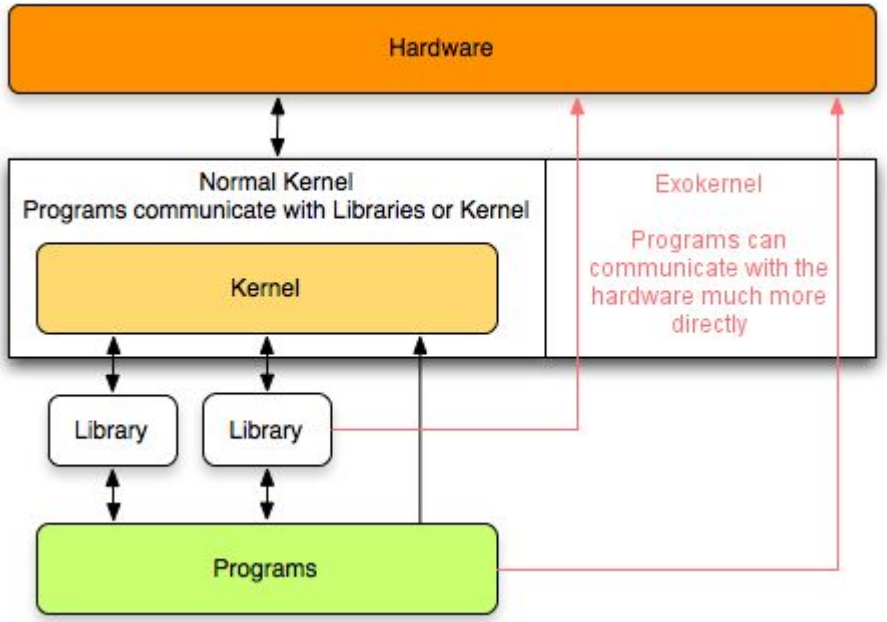
- Уменьшение производительности.


Клиент-серверная архитектура




Передача информации требует больших расходов и большого количества времени

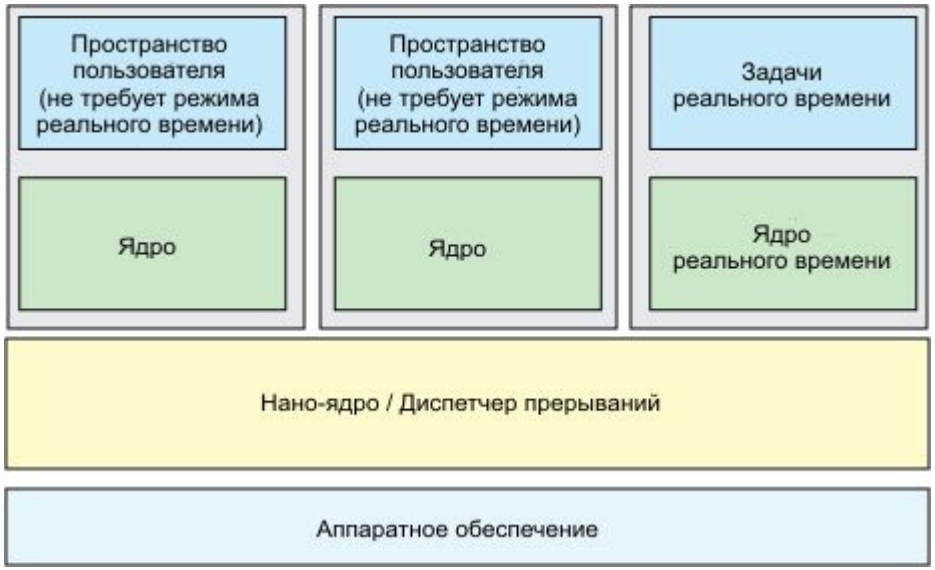
# Экзоядро




Такое ядро ОС, которое предоставляет лишь функции взаимодействия процессов, безопасное выделение и распределение ресурсов. Доступ к устройствам на уровне контроллеров позволяет решать задачи, которые нехарактерны для универсальной ОС

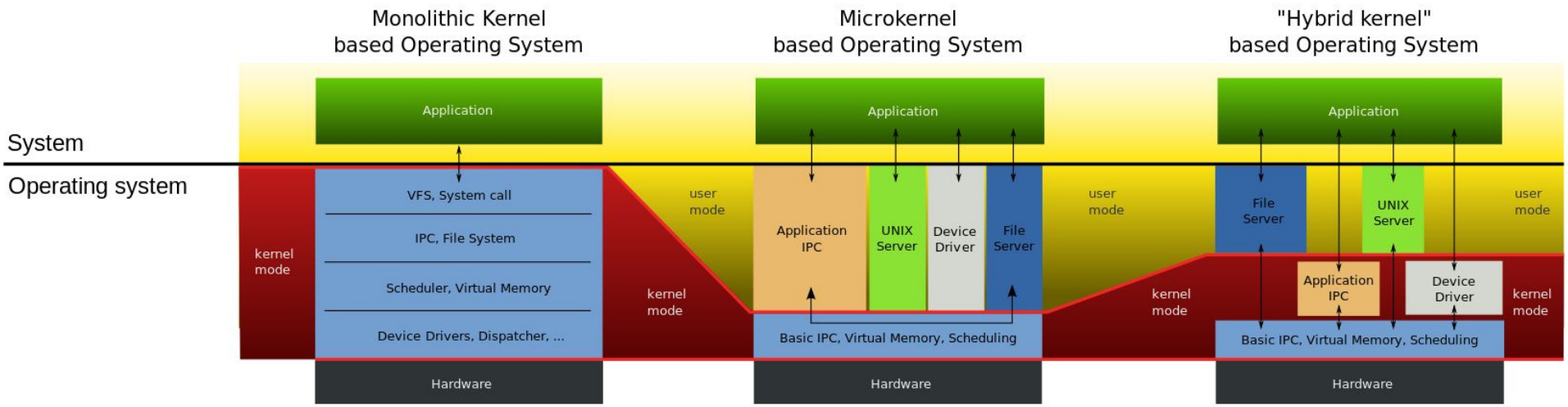
Принцип экзоядра, все отдать пользовательским программам. Например, зачем нужна файловая система? Почему не позволить пользователю просто читать и писать участки диска защищенным образом? Т.е. каждая пользовательская программа сможет иметь свою файловую систему. Такая операционная система должна обеспечить безопасное распределение ресурсов среди соревнующихся за них пользователей.

# Наноядро




Такое ядро выполняет только единственную задачу- обработку аппаратных прерываний, образуемых устройствами ПК. После обработки наноядро посылает данные о результатах обработки далее идущему в цепи программному обеспечения при помощи той же системы прерываний.

# Гибридное ядро




Модификация микроядра, позволяющая для ускорения позволяющие для ускорения работы запускать "несущественные" части в пространстве ядра. На архитектуре гибкого ядра построены последние операционные системы от Windows, в том числе и Windows 7.

Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерных операционных системах. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром

# Классификация операционных систем


По числу одновременно работающих пользователей ОС делятся: на однопользовательские (MS-DOS, Windows 3x, ранние версии OS/2) и многопользовательские (UNIX, Windows NT/2000/2003/XP/Vista).

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.

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).

Важное свойство ОС – отсутствие или наличие средств поддержки многопроцессорной обработки. По этому признаку можно выделить ОС без поддержки мультипроцессированя и (Windows 3.x, Windows 95) и с поддержкой мультипроцессирования (Solaris, OS/2, UNIX, Windows NT/2000/2003/XP/7/8/10)
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).

# Интерфейс прикладного программирования

**API (Application Programming Interface)** - интерфейс прикладного программирования.
Интерфейс между операционной системой и программами определяется набором **системных вызовов**.
Из наиболее часто применяемых системных вызовов - стандарта POSIX (более 100 системных вызовов).

- **fork** - создание нового процесса
- **exit** - завершение процесса
- **open** - открывает файл
- **close** - закрывает файл
- **read** - читает данные из файла в буфер
- **write** - пишет данные из буфера в файл
- **stat** - получает информацию о состоянии файла
- **mkdir** - создает новый каталог
- **rmdir** - удаляет каталог
- **link** - создает ссылку
- **unlink** - удаляет ссылку
- **mount** - монтирует файловую систему
- **umount** - демонтирует файловую систему
- **chdir** - изменяет рабочий каталог

В UNIX вызовы почти один к одному идентичны библиотечным процедурам, которые используются для обращения к системным вызовам

Рассмотрим вызовы Win32 API, которые подобны вызовам стандарта POSIX. Win32 API отделен от системных вызовов

- **CreatProcess** (fork) - создание нового процесса
- **ExitProcess** (exit) - завершение процесса
- **CreatFile** (open) - открывает файл
- **CloseHandle** (close) - закрывает файл
- **ReadFile** (read) - читает данные из файла в буфер
- **WriteFile** (write) - пишет данные из буфера в файл
- **CreatDirectory** (mkdir) - создает новый каталог
- **RemoveDirectory** (rmdir) - удаляет каталог
- **SetCurrentDirectory** (chdir) - изменяет рабочий каталог

В Win32 API существует более 1000 вызовов. Такое количество связано и с тем, что графический интерфейс пользователя UNIX запускается в пользовательском режиме, а в Windows встроен в ядро. Поэтому Win32 API имеет много вызовов для управления окнами, текстом, шрифтами т.д.

# Процессы

**Процесс** (задача) - программа, находящаяся в режиме выполнения.

С каждым процессом связывается его **адресное пространство,** из которого он может читать и в которое он может писать данные.

**Адресное пространство** содержит:

- саму программу (код)
- данные к программе
- стек программы

С каждым процессом связывается набор **регистров,** например:

- счетчика команд (в процессоре) - регистр в котором содержится адрес следующей, стоящей в очереди на выполнение команды. После того как команда выбрана из памяти, счетчик команд корректируется и указатель переходит к следующей команде.
- указатель стека
- и д.р.

Некоторые поля таблицы:

| Управление процессом | Управление памятью | Управление файлами |
|---|---|---|
| Регистры<br>Счетчик команд<br>Указатель стека<br>Состояние процесса<br>Приоритет<br>Параметры планирования<br>Идентификатор процесса<br>Родительский процесс<br>Группа процесса<br>Время начала процесса<br>Использованное процессорное время | Указатель на текстовый сегмент<br>Указатель на сегмент данных<br>Указатель на сегмент стека | Корневой каталог<br>Рабочий каталог<br>Дескрипторы файла<br>Идентификатор пользователя<br>Идентификатор группы |

Во многих операционных системах вся информация о каждом процессе, дополнительная к содержимому его собственного адресного пространства, хранится в **таблице процессов** операционной системы.

## Создание процесса

Три основных события, приводящие к созданию процессов (вызов **fork** или **CreateProcess**):

- Загрузка системы
- Работающий процесс подает системный вызов на создание процесса
- Запрос пользователя на создание процесса

Во всех случаях, активный текущий процесс посылает системный вызов на создание нового процесса.
В UNIX каждому процессу присваивается идентификатор процесса ( PID - Process IDentifier)

## Завершение процесса

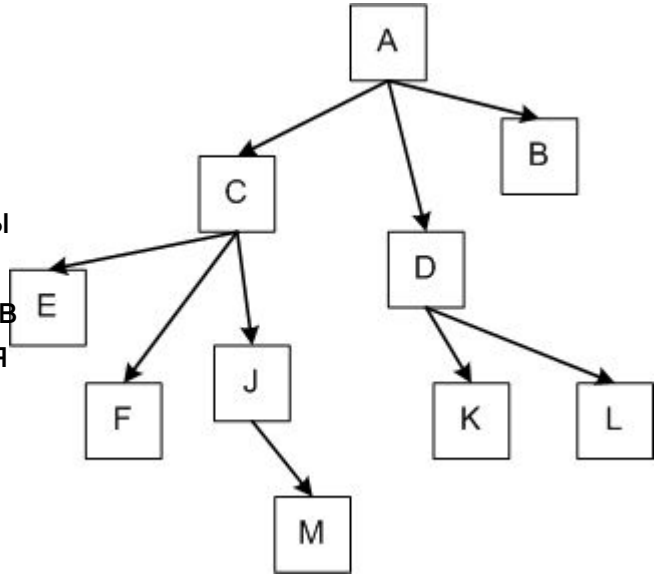
Четыре события, приводящие к остановке процесса (вызов **exit** или **ExitProcess**):

- Плановое завершение (окончание выполнения)
- Плановый выход по известной ошибке (например, отсутствие файла)
- Выход по неисправимой ошибке (ошибка в программе)
- Уничтожение другим процессом

Таким образом, приостановленный процесс состоит из собственного адресного пространства, обычно называемого **образом памяти** (**core image**), и компонентов таблицы процессов (в числе компонентов и его регистры).

## Иерархия процессов

В UNIX системах заложена жесткая иерархия процессов.
Каждый новый процесс созданный системным вызовом fork, является дочерним к предыдущему процессу. Дочернему процессу достаются от родительского переменные, регистры и т.п. После вызова fork, как только родительские данные скопированы, последующие изменения в одном из процессов не влияют на другой, но процессы помнят о том, кто является родительским.
В таком случае в UNIX существует и прародитель всех процессов - процесс **init**.



# Состояние процессов

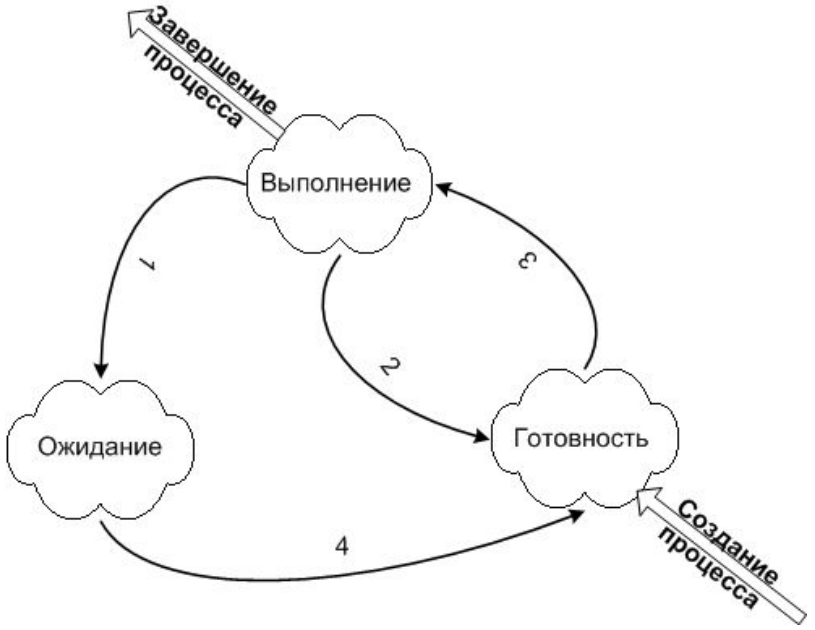
Три состояния процесса:

- Выполнение (занимает процессор)
- Готовность (процесс временно приостановлен, чтобы позволить выполняться другому процессу)
- Ожидание (процесс не может быть запущен по своим внутренним причинам, например, ожидая операции ввода/вывода)

Возможные переходы между состояниями.

1. Процесс блокируется, ожидая входных данных
2. Планировщик выбирает другой процесс
3. Планировщик выбирает этот процесс
4. Поступили входные данные

Переходы 2 и 3 вызываются планировщиком процессов операционной системы, так что сами процессы даже не знают о этих переходах. С точки зрения самих процессов есть два состояния выполнения и ожидания.

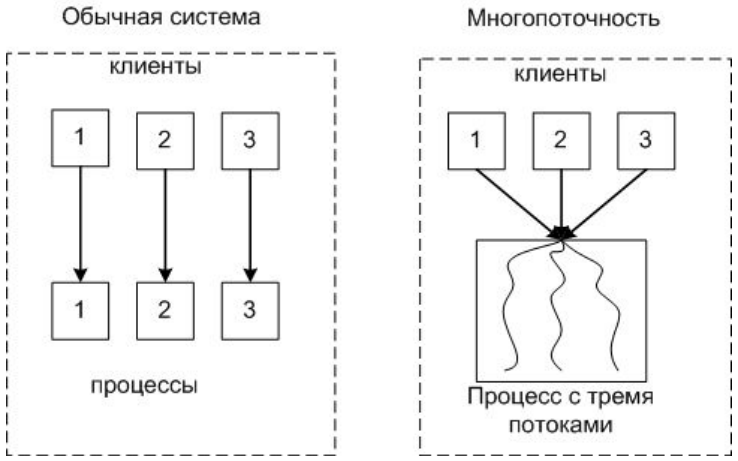



На серверах для ускорения ответа на запрос клиента, часто загружают несколько процессов в режим ожидания, и как только сервер получит запрос, процесс переходит из "ожидания" в "выполнение". Этот переход выполняется намного быстрее, чем запуск нового процесса.

# Потоки (нити, облегченный процесс)

## Понятие потока

Каждому процессу соответствует адресное пространство и одиночный **поток** исполняемых команд. В многопользовательских системах, при каждом обращении к одному и тому же сервису, приходится создавать новый процесс для обслуживания клиента. Это менее выгодно, чем создать квазипараллельный поток внутри этого процесса с одним адресным пространством.




Сравнение многопоточной системы с однопоточной

## Модель потока

С каждым потоком связывается:

- Счетчик выполнения команд
- Регистры для текущих переменных
- Стек
- Состояние

Потоки делят между собой элементы своего процесса:

- Адресное пространство
- Глобальные переменные
- Открытые файлы
- Таймеры
- Семафоры
- Статистическую информацию.

В остальном модель идентична модели процессов.

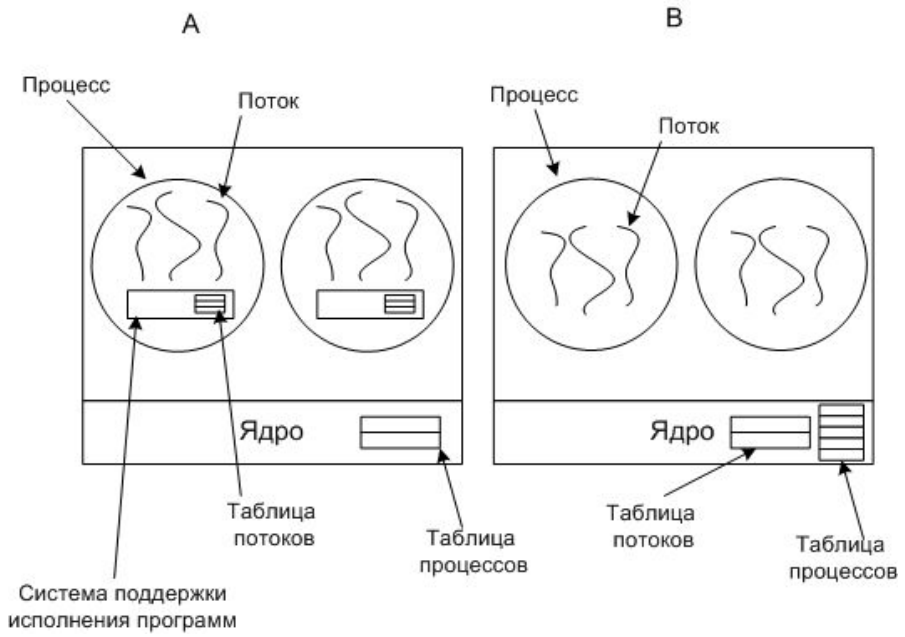
В POSIX и Windows есть поддержка потоков на уровне ядра.

# Реализация потоков в пространстве пользователя, ядра и смешанное

**A** - потоки в пространстве пользователя (модель N:1)

Примеры реализации:

1. GNU Portable Threads (реализация Posix Threads в пользовательском режиме. )
2. Carbon Threads




**B** - потоки в пространстве ядра (классическая модель 1:1)

Примеры реализации:

1. Потоки Win32 (Win32 API Threads).
2. Реализация Posix Threads в Linux (Native Posix Threads Library (NPTL)).
3. Легковесные ядерные потоки (Light Weight Kernel Threads — LWKT)

В случае **A** ядро о пользовательских потоках ничего не знает. Каждому процессу необходима **таблица потоков**, аналогичная таблице процессов.

Преимущества случая **A**:

- Такую многопоточность можно реализовать на ядре не поддерживающим многопоточность
- Более быстрое переключение, создание и завершение потоков
- Процесс может иметь собственный алгоритм планирования.

Недостатки случая **A**:

- При использовании блокирующего (процесс переводится в режим ожидания, например: чтение с клавиатуры, а данные не поступают) системного запроса все остальные потоки блокируются.
- Отсутствие прерывания по таймеру внутри одного процесса
- Сложность реализации

# Взаимодействие между процессами и потоками

## Взаимодействие между процессами

Ситуации, когда приходится процессам взаимодействовать:

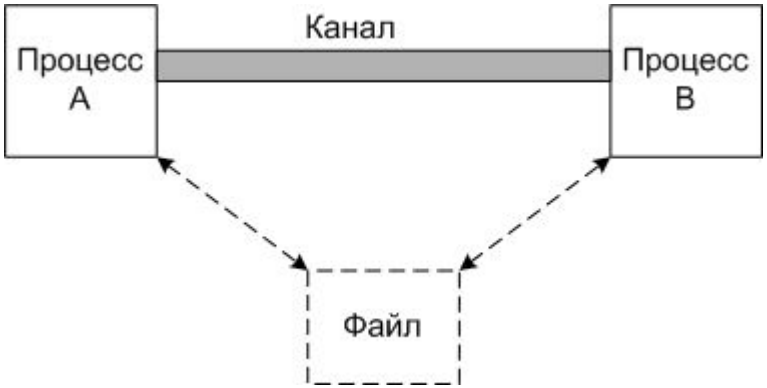
- Передача информации от одного процесса другому
- Контроль над деятельностью процессов (например: когда они борются за один ресурс)
- Согласование действий процессов (например: когда один процесс поставляет данные, а другой их выводит на печать. Если согласованности не будет, то второй процесс может начать печать раньше, чем поступят данные).

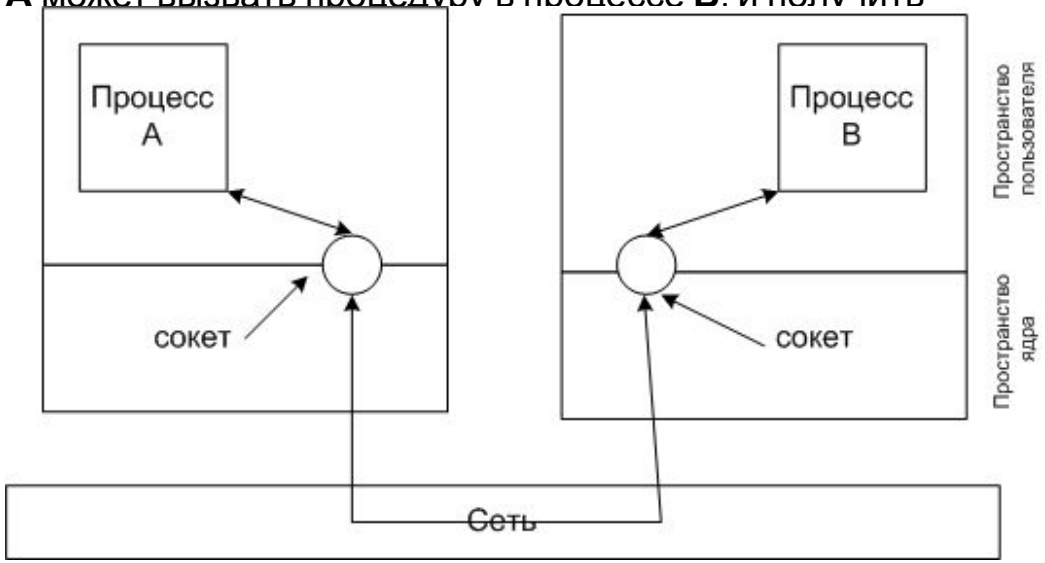
  **Два вторых случая относятся и к потокам.** В первом случае у потоков нет проблем, т.к. они используют общее адресное пространство.

## Передача информации от одного процесса другому

Передача может осуществляться несколькими способами:

- **Разделяемая память**
- **Каналы** (трубы), это псевдофайл, в который один процесс пишет, а другой читает.
- **Сокеты** - поддерживаемый ядром механизм, скрывающий особенности среды и позволяющий единообразно взаимодействовать процессам, как на одном компьютере, так и в сети.
- **Почтовые ящики** (только в Windows), однонаправленные, возможность широковещательной рассылки.
- **Вызов удаленной процедуры**, процесс **А** может вызвать процедуру в процессе **В**, и получить обратно данные.



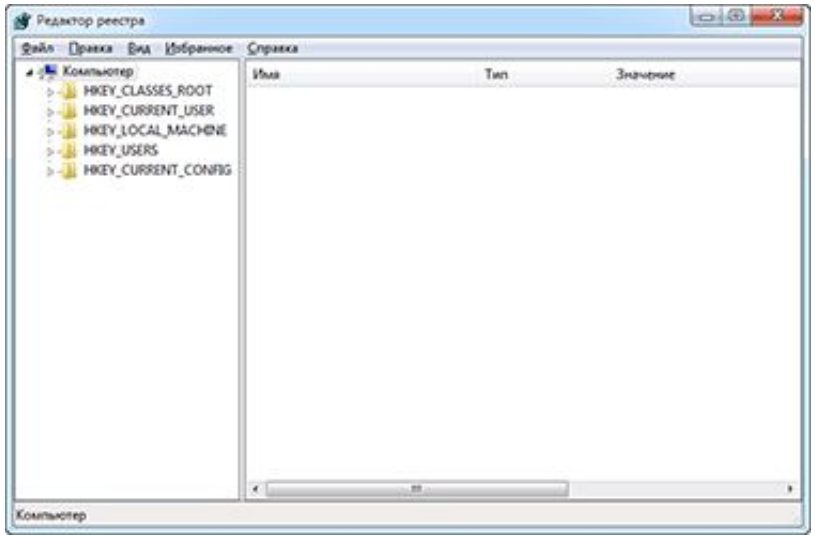



# Реестр Windows

**Реестр Windows** (англ. *Windows Registry*), или **системный реестр** — иерархически построенная база данных параметров и настроек в большинстве операционных систем Microsoft Windows.

Реестр содержит информацию и настройки для аппаратного обеспечения, программного обеспечения, профилей пользователей, предустановки. Большинство изменений в Панели управления, ассоциации файлов, системные политики, список установленного ПО фиксируются в реестре.

Реестр Windows был введён для упорядочения информации, хранившейся до этого во множестве INI-файлов, обеспечения единого механизма (API) записи-чтения настроек и избавления от проблем коротких имён, отсутствия разграничения прав доступа и медленного доступа к ini-файлам, хранящимся на файловой системе FAT16. Со временем (с появлением NTFS) проблемы, решавшиеся реестром, исчезли, но реестр остался из-за обратной совместимости, и присутствует во всех версиях Windows, включая последнюю. Поскольку сейчас не существует реальных предпосылок для использования подобного механизма, Microsoft Windows — единственная операционная система из используемых сегодня, в которой используется механизм реестра операционной системы.




В реестре есть пять базовых разделов (кустов) служащих для определенных задач:

- HKEY_USERS (HKU) — хранит настройки всех профилей пользователей имеющихся в компьютере.
- HKEY_LOCAL_MACHINE (HKLM) — служит для сохранения параметров конфигурации для всего компьютера.
- HKEY_CURRENT_CONFIG (HKCC) — хранит перечень устройств, использующихся при загрузке компьютера.
- HKEY_CURRENT_USER (HKCU) — является подразделом HKEY_USERS и отвечает за сохранение всех настроек учетной записи пользователя, который сейчас загружен в Windows.
- HKEY_CLASSES_ROOT (HKCR) — хранит информацию обо всех зарегистрированных в операционной системе типах файлов и их ассоциациях с программами. Включает в себя данные из подразделов HKEY_LOCAL_MACHINE\Software\Classes и

# Командная строка

**Интерфейс командной строки** (англ. Command line interface, CLI) — разновидность текстового интерфейса (TUI) между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «**консоль**» и «**терминал**».

**Примеры:**

1. **Win**

   **Запуск Яндекс-браузера из командной строки**:

   1. Через хоткей ***Win+R*** выполнить команду: cmd + ***<Enter>***
   2. Перейти в папку приложения с помощью команды: cd «путь» ( стандартный путь для Яндекс браузера: C:\Users\reveg\AppData\Local\Yandex\YandexBrowser\Application )
   3. Консоль поддерживает автоматическое дополнение имя файла, поэтому введя нчальные буквы файла можно переключаться между имеющимися с помощью Tab-ляции (введем: bro + ***<Tab>***)
   4. После имени файла вводим ключ эксперимента (например: --force-fieldtrials=nsw/1)
   5. Нажимаем ***<Enter>***

2. **Linux**

   **Установка Яндекс-браузера:**

   1. Открыть Терминал.
   2. Ввести: sudo apt install yandex*.deb
   3. Нажимаем ***<Enter>***

   **Запуск Яндекс-браузера:**

   1. Открыть Терминал.
   2. Ввести: yandex + ***<Tab>***
   3. Вводим ключ эксперимента (например: --force-fieldtrials=nsw/1 )
   4. Нажимаем ***<Enter>***

3. **MacOS**

   **Запуск Яндекс-браузера:**

   1. Открыть Терминал.
   2. Ввести: open yandex.app --args --force-fieldtrials=nsw/1
   3. Нажимаем ***<Enter>***

Любая операционная система имеет набор диагностических утилит для тестирования сетевых настроек и функционирования коммуникаций. Большой набор диагностических средств есть и в системах семейства Windows (как графических, так и режиме командной строки).
Перечислим утилиты командной строки, являющиеся инструментами первой необходимости для проверки настроек протокола TCP/IP и работы сетей и коммуникаций. Подробное описание данных утилит содержится в системе интерактивной помощи Windows.
В Таблице 6 укажем основные и наиболее часто используемые параметры этих команд и дадим их краткое описание.

| Название утилиты | Параметры | Комментарии |
|---|---|---|
| ipconfig | /? - Отобразить справку по команде<br>/all – Отобразить информацию о настройке параметров всех адаптеров<br>/release - Освободить динамическую IP-конфигурацию<br>/*renew* - Обновить динамическую IP-конфигурацию с DHCP-сервера<br>/flushdns - Очистить кэш разрешений DNS<br>/registerdns - Обновить регистрацию на DNS-сервере<br>/displaydns - Отобразить содержимое кэша разрешений DNS | Служит для отображения всех текущих параметров сети TCP/IP и обновления параметров DHCP и DNS. При вызове команды ipconfig без параметров выводятся IP-адрес, маска подсети и основной шлюз для каждого сетевого адаптера. |
| arp | -a - Отображает текущие ARP-записи | Отображение и изменение ARP-таблиц. |
| ping | Формат команды: "ping <сетевой узел> параметры" Параметры:<br>-t - Бесконечная (до нажатия клавиш <Ctrl>+<Break>) отправка пакетов на указанный узел<br>-a - Определение имени узла по IP-адресу<br>-n <число> - Число отправляемыхзапросов<br>-l <размер> - Размер буфера отправки<br>-w <таймаут> - Таймаут ожидания каждого ответа в миллисекундах | •Мощный инструмент диагностики (с помощью протокола ICMP). Команда ping позволяет проверить:работоспособность IP-соединения;<br>•правильность настройки протокола TCP/IP на узле;<br>•работоспособность маршрутизаторов;<br>•работоспособность системы разрешения имен FQDN или NetBIOS;<br>•доступность и работоспособность какого-либо сетевого ресурса. |
| tracert | -d - Без разрешения IP-адресов в имена узлов<br>-h <максЧисло> - Максимальное число прыжков при поиске узла<br>-w <таймаут> - Таймаут каждого ответа в миллисекундах | Служебная программа для трассировки маршрутов, используемая для определения пути, по которому IP-дейтаграмма доставляется по месту назначения. |
| pathping | -n - Без разрешения IP-адресов в имена узлов<br>-h максЧисло - Максимальное число прыжков при поиске узла<br>-q <число_запросов> - Число запросов при каждом прыжке<br>-w <таймаут> - Таймаут каждого ответа в миллисекундах | Средство трассировки маршрута, сочетающее функции программ ping и tracert и обладающее дополнительными возможностями. Эта команда показывает степень потери пакетов на любом маршрутизаторе или канале, с ее помощью легко определить, какие маршрутизаторы или каналы вызывают неполадки в работе сети. |
| netstat | -a - Отображение всех подключений и ожидающих (слушающих) портов<br>-n - Отображение адресов и номеров портов в числовом формате<br>-o - Отображение кода (ID) процесса каждого подключения<br>-r - Отображение содержимого локальной таблицы маршрутов | Используется для отображения статистики протокола и текущих TCP/IP-соединений. |
| nbtstat | –n - Выводит имена пространства имен NetBIOS, зарегистрированные*локальными процессами*<br>–c - Отображает кэш имен NetBIOS (разрешение NetBIOS-имен в IP- | Средство диагностики разрешения имен NetBIOS |

**Прокси-сервер** *(от англ. proxy — право пользоваться от чужого имени)* — удаленный компьютер, который, при подключении к нему вашей машины, становится посредником для выхода абонента в интернет. Прокси передает все запросы программ абонента в сеть, и, получив ответ, отправляет его обратно абоненту.

- **Анонимность**
- **Смена геолокции**
- **Кэширование трафика**
- **Анализ трафика (Fiddler, Charles, Wireshark)**


Взаимодействие веб-браузера клиента и WEB-сервера с использованием протокола HTTPS выглядит следующим образом:

1. Веб-браузер клиента запрашивает у WEB-сервера страницу используя метод GET по HTTPS протоколу, например:

https://gmail.com

2. В ответ WEB-сервер отправляет веб-браузеру клиента открытый ключ и свой сертификат.

3. Веб-браузер клиента выполняет проверку сертификата по сроку действия, отзыву сертификата, используя протокол OCSP (Online Certificate Status Protocol) и список отозванных сертификатов CRL (Certificate Revocation List), доверию сертификату сервера, совпадению значению поля Common Name в сертификате сервера с запрашиваемым доменным именем.

4. В случае прохождения проверок веб-браузер генерирует симметричный ключ, шифрует его открытым ключом WEB-сервера и передает WEB-серверу.

5. WEB-сервер расшифровывает симметричный ключ используя свой закрытый ключ.

6. Далее WEB-сервер отправляет WEB-страницу зашифрованную симметричным ключом.

7. Веб-браузер клиента расшифровывает содержимое WEB-страницы используя симметричный ключ и отображает содержимое.

# Особенности тестирования desktop приложений

**Функциональное тестирование** — это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности ПО в определённых условиях решать задачи, нужные пользователям. Функциональные требования определяют, что именно делает ПО, какие задачи оно позволяет.

**Юзабилити-тестирование** — исследование, выполняемое с целью определения, удобен ли некоторый искусственный объект (такой как веб-страница, пользовательский интерфейс или устройство) для его предполагаемого применения

**Тестирование графического интерфейса пользователя (GUI)** является обнаружение ошибок следующего характера: Ошибки в функциональности посредством интерфейса

- Необработанные исключения при взаимодействии с интерфейсом
- Потеря или искажение данных, передаваемых через элементы интерфейса
- Ошибки в интерфейсе (несоответствие проектной документации, отсутствие элементов интерфейса)

**Виды тестирования которые необходимо проводить на десктопных приложениях помимо основных:**

Выполняя **тестирование установки** проверяется:

- Запускается ли программа после установки
- Расположение программы в файловой системе по-умолчанию
- Расположение программы в файловой системе если путь сохранения изменен пользователем
- Наличие ярлыков на рабочем столе
- Есть ли установленный компонент в меню Пуск > Программы
- При установке обратить внимание на издателя
- Установка программы для текущего пользователя/для всех пользователей компьютера
- Установка пользователем с правами админа
- Установка пользователем без прав админа

Выполняя **тестирование обновлений** нужно:

- Проверить что после установки обновлений данные пользователя не были повреждены
- Проверить что все созданные ранее пользователем файлы остались доступными

Выполняя **тестирование удаления** проверяем:

- Файлы должны удалиться
- Ярлык с рабочего стола исчез
- Удалена ли запись из меню Пуск > Все программы
- Убеждаемся, что нет папок с названием программы в личной папке текущего пользователя.