Эволюция и архитектура ОС.

1.1. История ОС.

Первые (1945-1955г.г.) компьютеры работали 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. Появились первые системы **пакетной обработки** (1955-1965г.г.), которые просто автоматизировали запуск одной программ за другой и тем самым увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем. Совокупность нескольких заданий, как правило, в виде колоды перфокарт, получила название пакета заданий.

Многозадачность (1965-1980) - это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько задач. Пока одна задача выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при последовательном выполнении задач , а выполняет другую задачу. Для этого создали систему распределения памяти, когда каждая задача загружается в свой участок оперативной памяти, называемый разделом.

Спулинг (spooling-подкачка) в то время задания считывались с перфокарт на диск в том темпе, в котором они появлялись в помещении вычислительного центра, а затем, когда

очередное задание завершалось, новое задание с диска загружалось в освободившийся раздел.

Системы разделения времени - вариант многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Это было сделано, чтобы каждый программист мог отлаживать свою программу в реальном времени. Фактически это была многопользовательская система. Естественно стали возникать проблемы защиты задач друг от друга.

В это время была разработана многопользовательская система MULTICS, которая должна была обеспечивать одновременную работу сотни пользователей.

В это время также стали бурно развиваться мини-компьютеры (первый был выпущен в 1961г.), на которые была перенесена система MULTICS. Эта работа в дальнейшем развилась в систему **UNIX.**

Появилось много разновидностей несовместимых UNIX, основные из них были System V и BSD. Чтобы было возможно писать программы, работающие в любой системе UNIX, был разработан стандарт **POSIX.** Стандарт POSIX определяет минимальный интерфейс системного вызова, который должны поддерживать системы UNUX.

В 1974г. был выпущен центральный процессор Intel 8080, для него была создана операционная система **CP/M.** В 1977г. она была переработана для других процессоров, например Z80.

В начале 80-х была разработана система **MS-DOS**, и стала основной системой для микрокомпьютеров.

В 80-х годах стало возможным реализовать **графический интерфейс пользователя** (GUI - Graphical User Interface), теория которого была разработана еще в 60-е годы. Первой реализовала GUI корпорация Macintosh.

С 1985 года выпустили **Windows**, в то время она была графической оболочкой к MS-DOS вплоть до 1995г., когда вышла **Windows 95.**

Уже тогда было ясно, что DOS с ее ограничениями по памяти и по возможностям файловой системы не может воспользоваться вычислительной мощностью появляющихся компьютеров. Поэтому IBM и Microsoft начинали совместно разрабатывать операционную систему **OS/2**. Она должна была поддерживать вытесняющую многозадачность, виртуальную память, графический пользовательский интерфейс, виртуальную машину для выполнения DOS-приложений. Первая версия вышла 1987г.

В дальнейшем Microsoft отошла от разработки OS/2, и стала разрабатывать **Windows NT.** Первая версия вышла в 1993г.

В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением сетевых или распределенных операционных систем.

Сетевая операционная система не имеет отличий от операционной системы однопроцессорного компьютера. Она обязательно содержит программную поддержку для

сетевых интерфейсных устройств (драйвер сетевого адаптера), а также средства для удаленного входа в другие компьютеры сети и средства доступа к удаленным файлам.

Распределенная операционная система, напротив, представляется пользователям простой системой, в которой пользователь не должен беспокоиться о том, где работают его программы или где расположены файлы, все это должно автоматически обрабатываться самой операционной системой.

В 1987г. была выпущена операционная система **MINIX**, она была построена на схеме **микро ядра**.

В 1991г. была выпущена **LINUX**, в отличии от микроядерной MINIX она стала монолитной.

Чуть позже вышла **FreeBSD** (основой для нее послужила BSD UNIX).

1.2. Назначение ОС.

Более подробная информация - http://ru.wikipedia.org/wiki/Операционная_система

1.2.1. ОС как виртуальная машина.

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

Например, чтобы считать или записать информацию на дискету, надо:

- Запустить двигатель вращения дискеты
- Управлять шаговым двигателем перемещения головки
- Следить за индикатором присутствия дискеты
- Выбрать номер блока на диске
- Выбрать дорожку
- Выбрать номер сектора на дорожке
- и т.д.

Все эти функции берет на себя операционная система.

1.2.2. 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, например, так система управляет процессором.
- **Пространственный** - программа получает часть ресурса, например, так система управляет оперативной памятью и жестким диском.

1.3. Интерфейс прикладного программирования.

Более подробная информация - http://ru.wikipedia.org/wiki/API

API (Application Programming Interface) - интерфейс прикладного программирования. Интерфейс между операционной системой и программами определяется набором **системных вызовов**.

Например, если пользовательскому процессу необходимо считать данные из файла, он должен выполнить команду системного вызова, т.е. выполнить прерывание с переключением в режим ядра и активизировать функцию операционной системы для считывания данных из файла.

1.3.1. UNIX/Linux API.

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

В POSIX существует более 100 системных вызовов. Например,

- **fork** - создание нового процесса
- **exit** - завершение процесса
- **open** - открывает файл
- **close** - закрывает файл

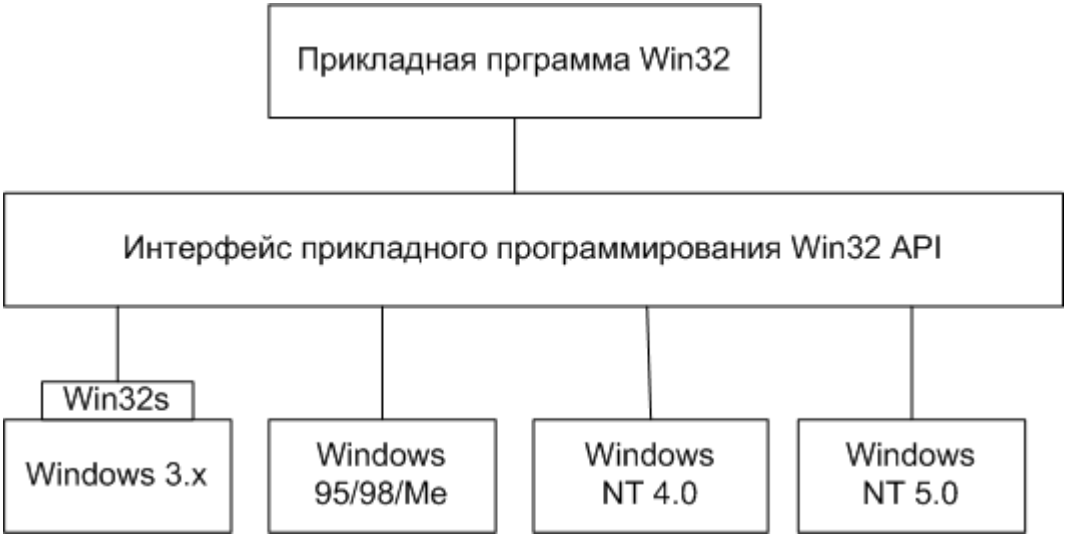
- **read** - читает данные из файла в буфер
- **write** - пишет данные из буфера в файл
- **stat** - получает информацию о состоянии файла
- **mkdir** - создает новый каталог
- **rmdir** - удаляет каталог
- **link** - создает ссылку
- **unlink** - удаляет ссылку
- **mount** - монтирует файловую систему
- **umount** - демонтирует файловую систему
- **chdir** - изменяет рабочий каталог

Более подробная информация - http://ru.wikipedia.org/wiki/POSIX

1.3.2. Windows API.

Интерфейс прикладного программирования для Windows - **Win32 API** отделен от системных вызовов. Это позволяет в разных версиях менять системные вызовы, не переписывая программы.

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.



Интерфейс Win32 API позволяет программам работать почти на всех версиях Windows

Рассмотрим вызовы Win32 API, которые подобны вызовам стандарта POSIX.

- **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

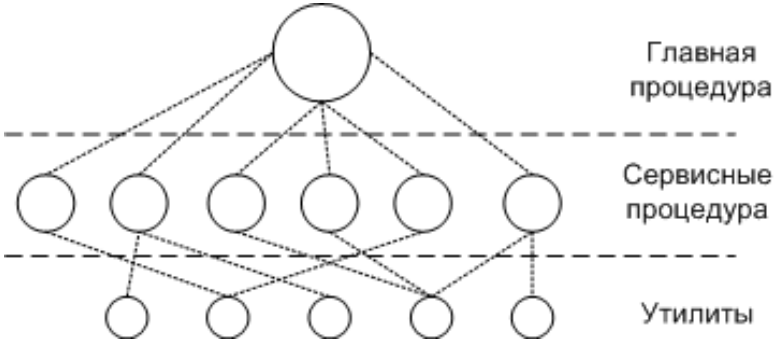
- **WriteFile** (write) - пишет данные из буфера в файл
- **CreatDirectory** (mkdir) - создает новый каталог
- **RemoveDirectory** (rmdir) - удаляет каталог
- **SetCurrentDirectory** (chdir) - изменяет рабочий каталог

Более подробная информация - http://ru.wikipedia.org/wiki/Windows_API

1.4. Структуры операционных систем.

1.4.1. Монолитная система.

1.4.1.1. Монолитное ядро — классическая и наиболее распространённая архитектура ядер операционных систем. Все части монолитного ядра работают в одном адресном пространстве.



Простая модель монолитной системы

Структура системы:

1. Главная программа, которая вызывает требуемые сервисные процедуры.
2. Набор сервисных процедур, реализующих системные вызовы.

3. Набор утилит, обслуживающих сервисные процедуры.

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.

Преимущества:

- Монолитные ядра имеют долгую историю развития и усовершенствования и, на данный момент, являются наиболее архитектурно зрелыми и пригодными к эксплуатации.

Недостатки:

- Монолитность ядер усложняет их отладку, понимание кода ядра, добавление новых функций и возможностей, удаление «мёртвого», ненужного, унаследованного от предыдущих версий кода.

- «Разбухание» кода монолитных ядер повышает требования к объёму оперативной памяти, требуемому для функционирования ядра ОС. Это делает монолитные ядерные архитектуры малопригодными к эксплуатации в системах, сильно ограниченных по объёму ОЗУ, например, встраиваемых системах, производственных микроконтроллерах и т. д.

1.4.1.2. Модульное ядро. Старые монолитные ядра требовали перекомпиляции при любом изменении состава оборудования. Большинство современных ядер, такие как OpenVMS, Linux, FreeBSD, MacOS X, позволяют во время работы динамически (по необходимости) подгружать и выгружать модули, выполняющие часть функций ядра. Модульность ядра осуществляется на уровне бинарного образа, а не на архитектурном уровне ядра, так как динамически подгружаемые модули загружаются в адресное пространство ядра и в дальнейшем работают как интегральная часть ядра.

Практически, динамическая загрузка модулей — это просто более гибкий способ изменения образа ядра во время выполнения — в отличие от перезагрузки с другим ядром.

Преимущества:

- Модули позволяют легко расширить возможности ядра по мере необходимости. Динамическая подгрузка модулей помогает сократить размер кода, работающего в пространстве ядра, до минимума, например, свести к минимуму размер ядра для встраиваемых устройств с ограниченными аппаратными ресурсами.

Загружаемый модуль ядра (англ. loadable kernel module, LKM) — объект, содержащий код, который расширяет функциональность запущенного или т. н. базового ядра ОС. Большинство текущих систем, основанных на Unix и Windows, поддерживают загружаемые модули ядра, хотя они могут называться по-разному (например, kernel loadable module в FreeBSD и kernel extension в Mac OS X).

Linux

Загружаемые модули ядра Linux загружаются с помощью команды modprobe. Обычно они находятся в каталоге /lib/modules и имеют расширение «.ko» («kernel object») для ядер с версии 2.6, и «.o» для ядер младших версий.

Mac OS X

Некоторые загружаемые модули ядра в Mac OS X могут быть загружены автоматически; они могут быть также загружены с помощью команды kextload. Модули, поставляемые с операционной системой, находятся в /System/Library/Extensions.

Windows

Ядро Windows само по себе не поддерживает расширения с помощью загружаемых модулей. Однако, поддерживаются загружаемые драйверы, а модуль, оформленный в виде драйвера Windows, не обязан работать с каким-либо внешним устройством. Благодаря этому, «псевдодрайверы» широко используются для изменения и расширения функциональности

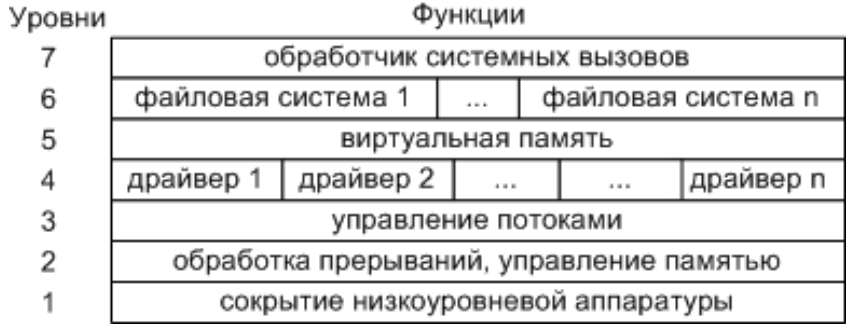
ядра Windows — анти-руткиты, перехватчики отладочного вывода, вспомогательные «агенты» многих системных программ, выпускаемых Sysinternals и т. п.

Безопасность

Хотя загружаемые модули ядра — это удобный метод модификации запущенного ядра, это также может быть использовано злоумышленником на скомпрометированной системе для скрытия его процессов и файлов, что позволяет ему сохранить контроль над системой. Многие руткиты используют загружаемые модули ядра подобным образом. Заметим, что с помощью модулей нельзя повысить полномочия, так как root-доступ все равно необходим для загрузки модуля; они лишь облегчают для злоумышленника пути сокрытия проникновения.

1.4.2. Многоуровневая структура ОС.

Развитием предыдущего подхода является организация ОС как иерархии уровней. Уровни образуются группами функций операционной системы - файловая система, управление процессами и устройствами и т.п. Каждый уровень может взаимодействовать только со своим непосредственным соседом - выше- или нижележащим уровнем. Прикладные программы или модули самой операционной системы передают запросы вверх и вниз по этим уровням.



Пример структуры многоуровневой системы

Преимущества:

- Высокая производительность

Недостатки:

- Большой код ядра, и как следствие большое содержание ошибок
- Ядро плохо защищено от вспомогательных процессов

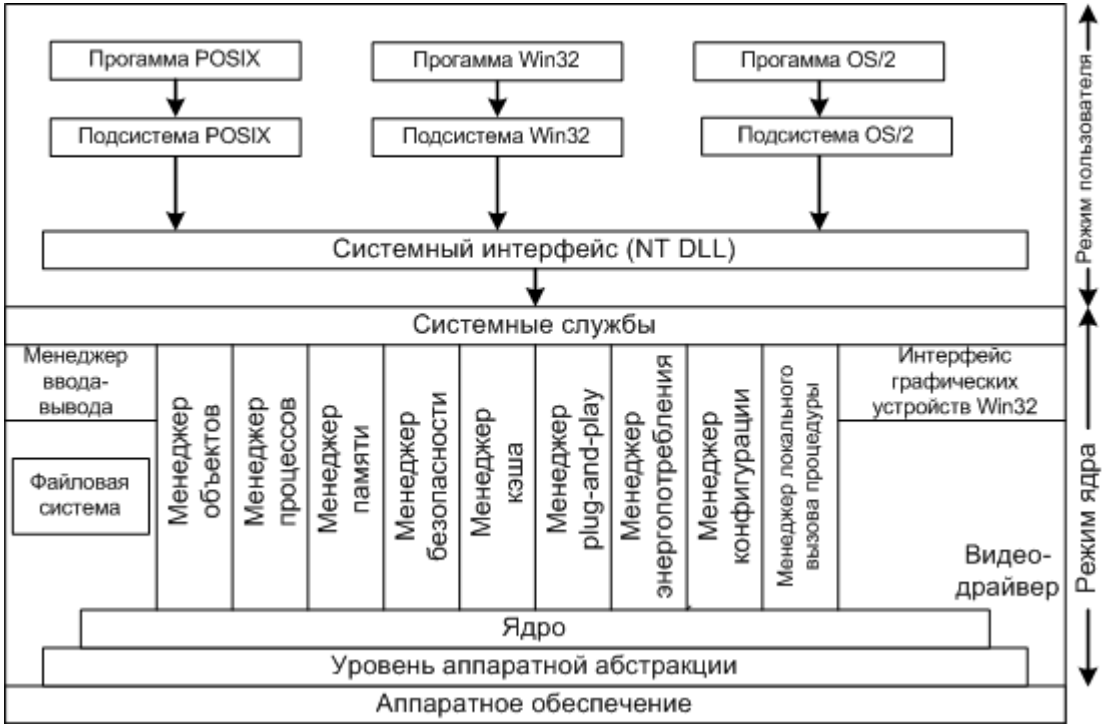
1.4.2.1. Пример реализации многоуровневой модели UNIX.

Структура ОС UNIX


Системные вызовы
Аппаратные и эмулированные прерывания
Управление терминалом
Сокеты
Именование файла
Отображение адресов
Страничные прерывания
Обработка сигналов
Создание и завершение процессов
Необработанный телетайп
Обработанный телетайп
Сетевые протоколы
Файловые системы
Виртуальная память
Дисциплины линии связи
Маршрутизация
Буферный кэш
Драйверы сетевых устройств
Планирование процесса
Символьные устройства
Драйверы сетевых устройств
Драйверы дисковых устройств
Диспетчеризация процессов
Аппаратура

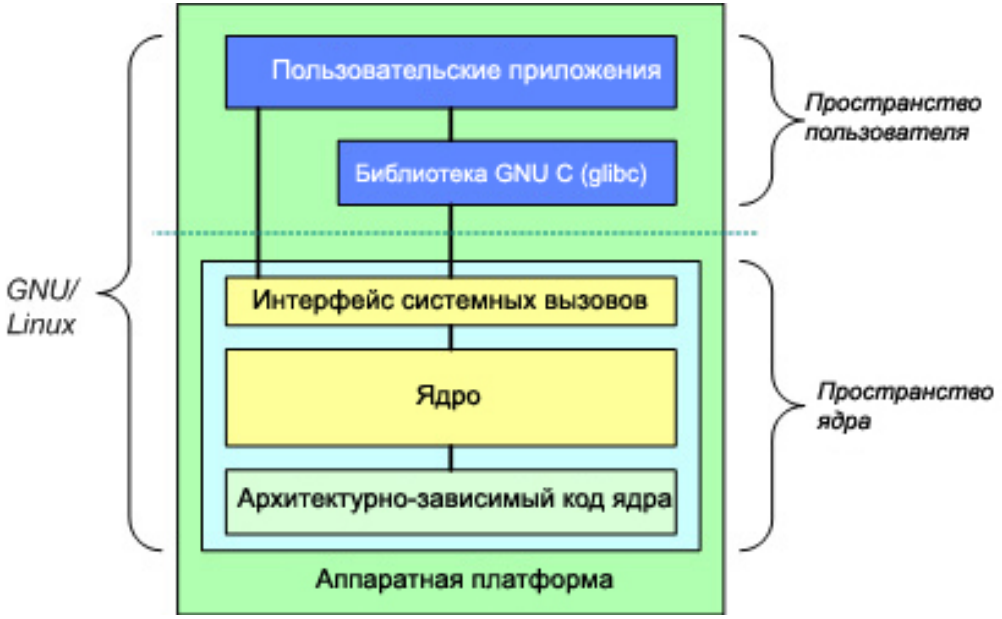
Ядро ОС UNIX

1.4.2.2. Пример реализации многоуровневой модели Windows.



Структура Windows 2000

1.4.2.3. Пример реализации многоуровневой модели Linux.



Фундаментальная архитектура операционной системы GNU/Linux

На верхнем уровне находится пользовательское пространство (пространство приложений). Здесь исполняются приложения пользователя. Под пользовательским пространством располагается пространство ядра. Здесь функционирует ядро Linux.

1.4.3. Модель экзоядра.

Более подробная информация - http://ru.wikipedia.org/wiki/Экзоядро

Если предыдущие модели брали на себя максимум функций, 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.

1.4.4. Микроядерная архитектура (модель клиент-сервер).

Более подробная информация - http://ru.wikipedia.org/wiki/Микроядро

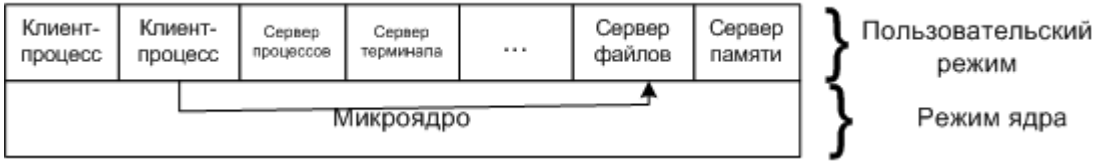
Эта модель является средним между двумя предыдущими моделями.

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.



Модель клиент-сервер

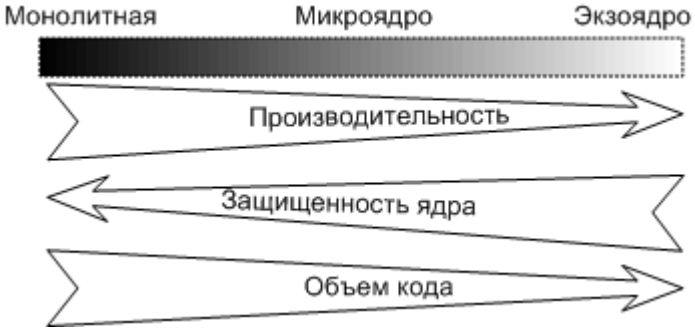
Преимущества:

- 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:

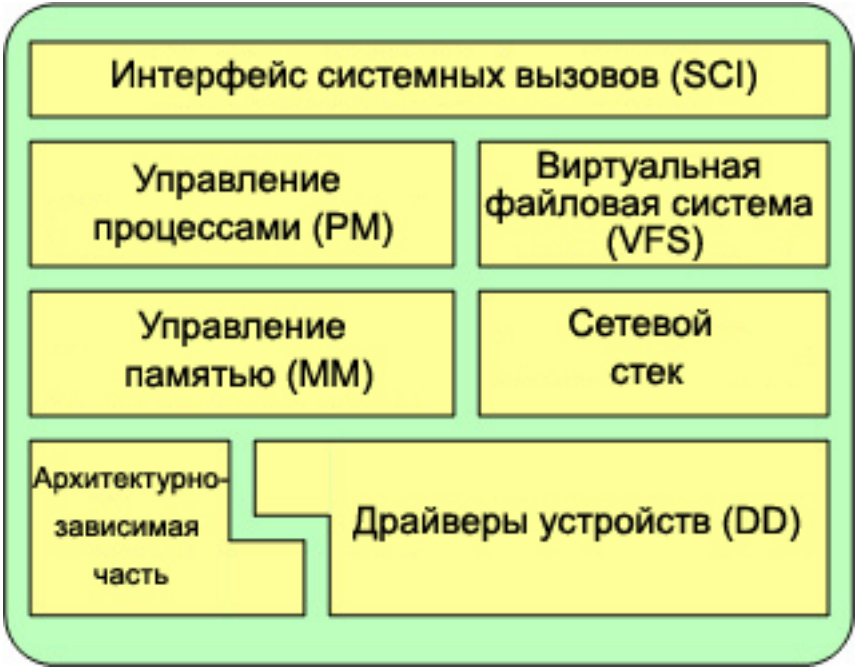
- Уменьшение производительности.

1.4.5. Обобщение сравнения моделей.



Сравнения моделей.

1.5. Основные подсистемы ядра Linux.



Основные компоненты ядра Linux

1.5.1. Интерфейс системных вызовов.

SCI - это уровень, предоставляющий средства для вызова функций ядра из пространства пользователя. Этот интерфейс может быть архитектурно зависимым, даже в пределах одного процессорного семейства. SCI фактически представляет собой службу мультиплексирования и демультиплексирования вызова функций. Реализация SCI находится в ./linux/kernel, а архитектурно-зависимая часть - в ./linux/arch.

1.5.2. Управление процессами.

В ядре эти процессы называются потоками (threads); они соответствуют отдельным виртуализованным объектам процессора (код потока, данные, стек, процессорные регистры). В пространстве пользователя обычно используется термин процесс, хотя в реализации Linux эти две концепции (процессы и потоки) не различают. Ядро предоставляет интерфейс программирования приложений (API) через SCI для создания нового процесса (порождения копии, запуска на исполнение, вызова функций [POSIX]), остановки процесса (kill, exit), взаимодействия и синхронизации между процессами (сигналы или механизмы POSIX).

Еще одна задача управления процессами - совместное использование процессора активными потоками. В ядре реализован новаторский алгоритм планировщика, время работы которого не зависит от числа потоков, претендующих на ресурсы процессора. Название этого планировщика - O(1) - подчеркивает, что на диспетчеризацию одного потока затрачивается столько же времени, как и на множество потоков. Планировщик O(1) также поддерживает симметричные многопроцессорные конфигурации (SMP). Исходные коды системы управления процессами находятся в ./linux/kernel, а коды архитектурно-зависимой части - в ./linux/arch).

1.5.3. Управление памятью.

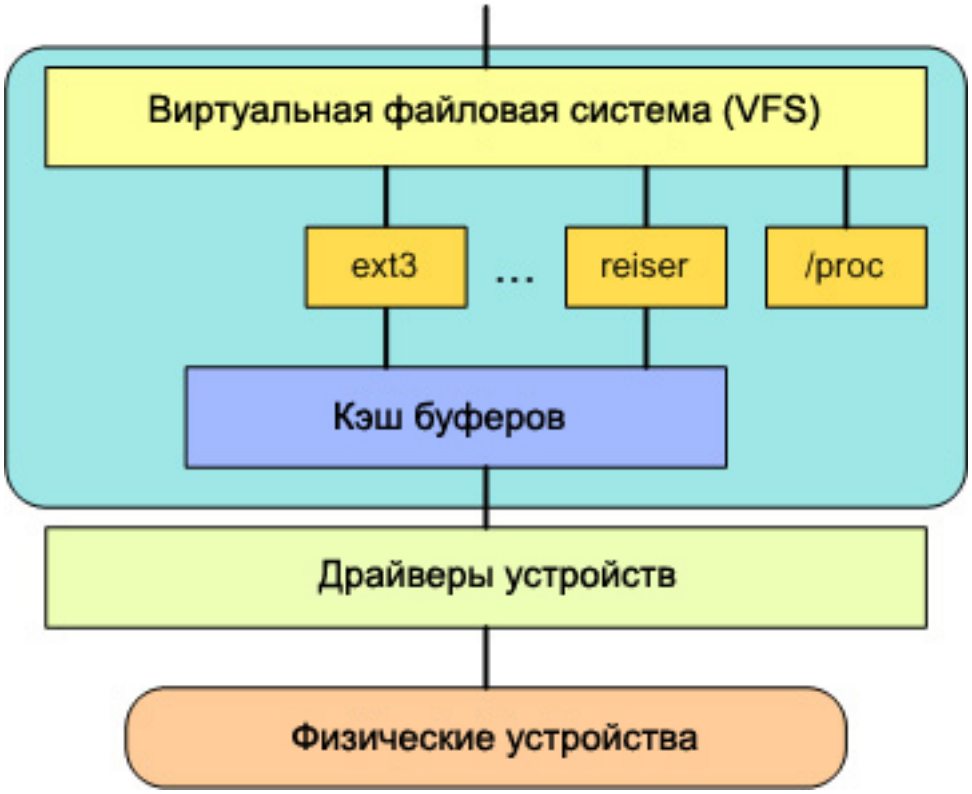
Другой важный ресурс, которым управляет ядро - это память. Для повышения эффективности, учитывая механизм работы аппаратных средств с виртуальной памятью, память организуется в виде т.н. страниц (в большинстве архитектур размером 4 КБ). В Linux имеются средства для управления имеющейся памятью, а также аппаратными механизмами для установления соответствия между физической и виртуальной памятью.

Однако управление памятью - это значительно больше, чем просто управление буферами по 4 КБ. Linux предоставляет абстракции над этими 4 КБ буферами, например, механизм распределения slab allocator. Этот механизм управления базируется на 4 КБ буферах, но затем размещает структуры внутри них, следя за тем, какие страницы полны, какие частично заполнены и какие пусты. Это позволяет динамически расширять и сокращать схему в зависимости от потребностей вышележащей системы.

В условиях наличия большого числа пользователей памяти возможны ситуации, когда вся имеющаяся память будет исчерпана. В связи с этим страницы можно удалять из памяти и переносить на диск. Этот процесс обмена страниц между оперативной памятью и жестким диском называется подкачкой. Исходные коды управления памятью находятся в ./linux/mm.

1.5.4. Виртуальная файловая система.

Еще один интересный аспект ядра Linux - виртуальная файловая система (VFS), которая предоставляет общую абстракцию интерфейса к файловым системам. VFS предоставляет уровень коммутации между SCI и файловыми системами, поддерживаемыми ядром (см. Рис.).



VFS предоставляет коммутационную матрицу между пользователями и файловыми системами

На верхнем уровне VFS располагается единая API-абстракция таких функций, как открытие, закрытие, чтение и запись файлов. На нижнем уровне VFS находятся абстракции файловых систем, которые определяют, как реализуются функции верхнего уровня. Они представляют собой подключаемые модули для конкретных файловых систем (которых существует более 50). Исходные коды файловых систем находятся в ./linux/fs.

Ниже уровня файловой системы находится кэш буферов, предоставляющий общий набор функций к уровню файловой системы (независимый от конкретной файловой системы). Этот уровень кэширования оптимизирует доступ к физическим устройствам за счет краткосрочного хранения данных (или упреждающего чтения, обеспечивающего готовность данных к тому моменту, когда они понадобятся). Ниже кэша буферов находятся драйверы устройств, реализующие интерфейсы для конкретных физических устройств.

1.5.5. Сетевой стек.

Сетевой стек по своей конструкции имеет многоуровневую архитектуру, повторяющую структуру самих протоколов. Вы помните, что протокол Internet Protocol (IP) - это базовый протокол сетевого уровня, располагающийся ниже транспортного протокола Transmission Control Protocol, TCP). Выше TCP находится уровень сокетов, вызываемый через SCI.

Уровень сокетов представляет собой стандартный API к сетевой подсистеме. Он предоставляет пользовательский интерфейс к различным сетевым протоколам. Уровень сокетов реализует стандартизованный способ управления соединениями и передачи данных между конечными точками, от доступа к "чистым" кадрам данных и блокам данных протокола

IP (PDU) и до протоколов TCP и User Datagram Protocol (UDP). Исходные коды сетевой подсистемы ядра находятся в каталоге ./linux/net.

1.5.6. Драйверы устройств.

Подавляющее большинство исходного кода ядра Linux приходится на драйверы устройств, обеспечивающие возможность работы с конкретными аппаратными устройствами. В дереве исходных кодов Linux имеется подкаталог драйверов, в котором, в свою очередь, имеются подкаталоги для различных типов поддерживаемых устройств, таких как Bluetooth, I2C, последовательные порты и т.д. Исходные коды драйверов устройств находятся в ./linux/drivers.

1.5.7. Архитектурно-зависимый код.

Хотя основная часть Linux независима от архитектуры, на которой работает операционная система, в некоторых элементах для обеспечения нормальной работы и повышения эффективности необходимо учитывать архитектуру. В подкаталоге ./linux/arch находится архитектурно-зависимая часть исходного кода ядра, разделенная на ряд подкаталогов, соответствующих конкретным архитектурам. Все эти каталоги в совокупности образуют BSP. В случае обычного настольного ПК используется каталог i386. Подкаталог для каждой архитектуры содержит ряд вложенных подкаталогов, относящихся к конкретным аспектам ядра, таким как загрузка, ядро, управление памятью и т.д. Исходные коды архитектурно-зависимой части находятся в ./linux/arch.