Учебник пурпурный

по дисциплине: Технологии системного программного обеспечения

Авторы:
Губин Егор ИУК4-62Б
Мосолов Алексей ИУК4-61Б
Иванов Роман ИУК4-62Б
Писаренко Артем ИУК4-61Б
Боков Артем ИУК4-62Б
Иванов Алексей ИУК4-61Б
Ткаченко Павел ИУК4-62Б

г. Калуга 2025 г.

Оглавление

1. Изложите концепцию системного программного обеспечения.

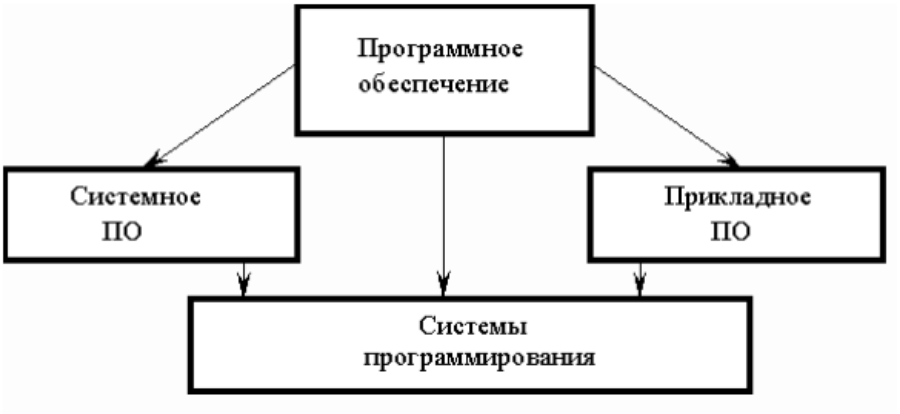

Рис. 1. Классификация программного обеспечения

Системное программное обеспечение — это совокупность программ, предназначенных для обеспечения функционирования вычислительной системы и управления её ресурсами. Оно служит основой, на которой работают все прикладные программы.

Основные компоненты системного ПО:

1. **Базовое ПО:**
   - Операционные системы (ОС);
   - Оболочки;
   - Сетевые операционные системы.
2. **Сервисное ПО:**
   - Утилиты для диагностики, антивирусной защиты, архивирования, обслуживания носителей и сетей.
3. **Системы программирования** (инструментальные средства) — используются для разработки новых программ:
   - Трансляторы, среды разработки, отладчики, редакторы связей, библиотеки.

Системное ПО управляет аппаратными ресурсами, предоставляет услуги прикладным программам и пользователю. Оно обеспечивает такие функции, как планирование процессов, управление памятью, файлами, устройствами ввода-вывода, сетевыми взаимодействиями.

Без СПО невозможна работа прикладных программ. Оно обеспечивает взаимодействие между пользователем, прикладными программами и аппаратным обеспечением, включая в себя как локальные, так и распределённые (сетевые) компоненты.

Так же к СПО относятся распределённые ОС, которые состоят из серверной части (предоставление ресурсов), клиентской части (доступ к удалённым ресурсам), коммуникационных средств (адресация, маршрутизация, буферизация сообщений и т. д.).

Рис. 2. Части операционной системы

2. Изложите концепцию синхронизации в распределенных системах.

Синхронизация в распределённых системах — это механизм, обеспечивающий согласованную работу процессов, взаимодействующих через передачу сообщений или вызовы удалённых процедур (RPC). Она необходима для организации совместного использования ресурсов, таких как файлы и устройства, а также для корректного обмена данными между удалёнными процессами.

В отличие от однопроцессорных систем, где синхронизация достигается с помощью семафоров, мониторов и других средств, использующих разделяемую память, в распределённых системах такие подходы неприменимы. Причина в том, что процессы могут выполняться на различных машинах и не иметь общего доступа к разделяемой оперативной памяти, в которой размещаются, например, семафоры (нужны новые механизмы синхронизации).

3. Перечислите и раскройте содержание основных этапов динамического связывания и семантики в случае отказов.

Этапы динамического связывания:

1. **Спецификация сервера**
   Начальным этапом динамического связывания является формальное описание сервера. Спецификация включает:
   - имя файл-сервера,
   - номер версии,
   - список процедур, предоставляемых сервером клиенту,
   - описание параметров процедур с указанием, какие из них являются входными, выходными или одновременно входными и выходными (например, массив, который

изменяется на сервере и возвращается клиенту).

2. **Связывание клиента с сервером**
   Вместо жёстко заданного сетевого адреса, клиент обращается к службе имен или таблице, которая по спецификации возвращает текущее местоположение сервера. Это обеспечивает гибкость при изменении числа серверов, их перемещении или обновлении интерфейса.
3. **Вызов удалённой процедуры (RPC)**
   После нахождения сервера клиент вызывает нужную процедуру, передавая параметры в соответствии с описанием из спецификации.

Семантика RPC в случае отказов:

1. **Неудачное определение местоположения сервера**
   Возникает, если сервер недоступен или устарела информация о его интерфейсе. Клиент получает сообщение об ошибке.
2. **Потеря запроса клиента**
   Решается повторной отправкой запроса через заданное время.
3. **Потеря ответа сервера**
   Проблема усложняется, если процедура не идемпотентна (т.е. при нескольких вызовах получаем один и тот же результат). Возможные решения:
   - сделать все процедуры идемпотентными
   - использовать нумерацию запросов, чтобы сервер мог определить, повторный ли это вызов, и не выполнять его повторно
4. **Авария сервера после получения запроса**
   Определить, был ли выполнен запрос, клиент не может. Если операция не идемпотентна, возможны риски изменения состояния. Здесь также важно учитывать момент сбоя — до или после выполнения процедуры.
5. **Авария клиента после отправки запроса**
   В этом случае сервер может выполнить операцию, но клиент уже не ждёт ответа. Такие "сиротские" вычисления могут вызывать:
   - ненужные затраты ресурсов
   - подмену ответов на другие запросы

   Возможные способы устранения сирот:
   - уничтожение
   - перевоплощение (привязка сироты к новому клиенту)
   - мягкое перевоплощение (временная привязка до подтверждения)
   - истечение срока (автоматическое завершение операций после таймаута)

4. Опишите интерфейсы системы UNIX и структуру её ядра. Поясните назначение основных утилит.

Интерфейсы системы UNIX

Операционная система UNIX имеет архитектуру, которую можно представить в виде пирамиды. В её основании находится аппаратное обеспечение — центральный процессор, память, диски, терминалы и другие устройства. Над «железом» работает ядро операционной системы UNIX, обеспечивая управление оборудованием и предоставляя программам интерфейс системных вызовов.

Программы обращаются к системным вызовам, передавая аргументы через регистры или стек и инициируя эмулированное прерывание для перехода в режим ядра. На языке C невозможно напрямую вызывать эмулированные прерывания, поэтому используются библиотечные функции, написанные на ассемблере. Именно они вызываются из C-программ (например, read()).

В стандарте POSIX описан интерфейс библиотечных функций, а не системных вызовов напрямую. Стандарт определяет, какие функции должны быть доступны, какие аргументы они принимают и какие результаты возвращают.
Три уровня интерфейсов:

1. Интерфейс системных вызовов — механизм взаимодействия программ с ядром.
2. Интерфейс библиотечных функций — набор функций на C, реализующих системные вызовы.
3. Интерфейс обслуживающих программ — стандартные утилиты, компиляторы, оболочки и другое

Структура ядра UNIX

Структура ядра системы UNIX сложна и может различаться в зависимости от версии. Она включает несколько уровней:

- Нижний уровень:
  - Драйверы устройств: делятся на символьные (без операции поиска) и блочные (поддерживают поиск). Сетевые устройства выделяются в отдельный класс
  - Диспетчеризация процессов: при прерывании текущее состояние сохраняется, запускается нужный драйвер
- Средние уровни:
  - Символьные устройства
  - Сетевое ПО: реализует маршрутизацию, стек протоколов (IP, TCP и др.) и интерфейс сокетов
  - Дисковая подсистема: включает буферный и страничный кэши, поддерживает различные файловые системы
  - Система виртуальной памяти: реализует отображение файлов в память, алгоритмы замещения страниц, обработку страничных прерываний
  - Процессное управление: включает планировщик процессов, обработку сигналов, создание и завершение процессов
- Верхний уровень:
  - Интерфейс системных вызовов — точка входа всех вызовов из пользовательского пространства
  - Обработка прерываний — аппаратные и программные прерывания, сигналы и исключения

Назначение основных утилит UNIX

UNIX содержит множество обслуживающих программ, называемых утилитами. Они делятся на шесть категорий:

1. Команды управления файлами и каталогами — создание, удаление, копирование, перемещение
2. Фильтры — обработки данных, например grep, sort, cut, awk
3. Средства разработки программ:
   - Компилятор cc — для языка C
   - Архиватор ar — собирает объектные файлы в библиотеки
   - Команда make — управляет сборкой больших программ, отслеживая зависимости между файлами и автоматизируя перекомпиляцию
4. Текстовые процессоры — работа с текстами и форматирование
5. Утилиты системного администрирования — управление пользователями, процессами, системой
6. Разное — другие инструменты, не попадающие в указанные категории

5. Изложите концепцию управления процессами в системе UNIX. Опишите специфику системных вызовов управления процессами и потоками.

Каждый процесс запускает одну программу и изначально получает один поток управления — то есть один счётчик команд, указывающий на следующую инструкцию. Изначально процесс имеет только один поток, большинство версий UNIX позволяют создавать дополнительные потоки во время выполнения.

В UNIX множество процессов могут выполняться одновременно. У каждого пользователя может быть запущено несколько процессов, а в целом в системе одновременно могут работать сотни и тысячи процессов. Многие из них — фоновые процессы, или демоны, запускаемые автоматически при старте системы.

Создание и управление процессами

Процессы создаются с помощью системного вызова fork(), который создаёт точную копию родительского процесса. Новый процесс называется дочерним. У каждого из процессов будет свой собственный адрес памяти, но открытые файлы, созданные до вызова fork, будут разделяться. Это обеспечивает совместную работу с файлами при сохранении независимости в памяти и переменных.

Системный вызов fork() возвращает 0 в дочернем процессе и PID дочернего процесса в родительском. Это позволяет программам различать роли процессов.

Дочерний процесс часто заменяет свой образ памяти с помощью системного вызова exec() для выполнения другой программы. Например, при вводе команды cp file1 file2, оболочка (shell) использует fork() для создания нового процесса, а затем exec() — для запуска команды копирования.

Для ожидания завершения дочернего процесса родительский использует вызов waitpid(), который может ожидать завершения конкретного дочернего процесса или любого, если задан PID -1. Возвращаемое значение позволяет узнать статус завершения дочернего процесса.

Если дочерний процесс завершается, а родительский не вызывает waitpid(), дочерний процесс переходит в состояние "зомби" до тех пор, пока родитель не соберёт его статус.

Завершение процессов и сигналы

Процесс завершает выполнение с помощью вызова exit(), возвращая родительскому процессу код завершения. Этот код затем можно получить через waitpid().

Общение между процессами осуществляется через **каналы (pipes)** и **сигналы**. Каналы позволяют одному процессу писать поток байтов, а другому — читать. Если канал пуст, процесс блокируется до появления данных.

Сигналы — это механизм уведомления процесса о каком-либо событии. Сигналы можно игнорировать, перехватывать или позволить завершить процесс (действие по умолчанию). Для обработки сигналов используется вызов sigaction(), а для отправки сигналов — kill(), несмотря на название, этот вызов часто используется не для уничтожения, а для управления процессами.

Управление потоками

Изначально каждая программа исполнялась в одном потоке. Добавление потоков позволило одному процессу иметь несколько точек исполнения — это улучшило производительность и позволило реализовать параллелизм внутри одного процесса. В UNIX-системах управление потоками реализуется с помощью библиотек POSIX Threads (pthreads).

6. Опишите особенности реализации процессов и потоков в UNIX. Раскройте механизм использования таблицы процессов и структуры пользователя.

Реализация процессов в UNIX

В UNIX ядро поддерживает две основные структуры данных, связанные с процессами: таблицу процессов и структуру пользователя.

1. **Таблица процессов**
   - Таблица процессов является резидентной (постоянно находящейся в памяти) и содержит информацию, необходимую для всех процессов системы, включая те, которые в данный момент не находятся в памяти. Это позволяет ядру отслеживать состояние всех процессов, даже если они не активны
   - Информация в таблице процессов включает:
     - **Параметры планирования**: приоритеты процессов, количество потребленного процессорного времени, время, проведенное в режиме ожидания, и другая информация, необходимая для выбора следующего процесса, которому будет передано управление
     - **Образ памяти**: указатели на сегменты кода, данных и стека, а также таблицы страниц, если используется страничная организация памяти. Если процесс выгружен в файл, то в таблице хранится информация о его расположении на диске
     - **Сигналы**: маски сигналов, которые указывают, какие сигналы игнорируются, какие перехватываются, блокируются или находятся в процессе доставки
     - **Разные данные**: текущее состояние процесса, идентификаторы пользователя и группы, PID родительского процесса и другие метаданные
2. **Структура пользователя**
   - Структура пользователя (или контекст процесса) содержит информацию, которая важна только в тот момент, когда процесс находится в памяти и выполняется
   - Когда процесс выгружается из памяти, структура пользователя может быть выгружена на диск, чтобы освободить место для других процессов
   - Данные, хранящиеся в структуре пользователя, включают:
     - **Машинные регистры**: сохраняются при переходе в режим ядра, включая регистры с плавающей точкой
     - **Состояние системного вызова**: включает информацию о текущем системном вызове, его параметрах и результатах
     - **Таблица дескрипторов файлов**: используется для нахождения структуры данных (i-node) файла, с которым работает процесс
     - **Учетная информация**: данные о процессорном времени, использованном процессом, а также другие ограничения
     - **Стек ядра**: фиксированная область памяти, используемая процессом при выполнении в режиме ядра

Создание нового процесса в UNIX

Когда пользователь запускает команду через оболочку, оболочка создаёт новый процесс. Это происходит с помощью системного вызова **fork**, который клонирует текущий процесс оболочки. Новый процесс (дочерний) затем вызывает **exec**, чтобы загрузить программу, например ls.

Процесс создания нового процесса выглядит следующим образом:

1. **Создание новой ячейки в таблице процессов**: для дочернего процесса создаётся новая запись, где копируется большинство данных из родительского процесса

2. **Настройка карты памяти**: дочернему процессу выделяются новые таблицы страниц, которые указывают на страницы памяти родительского процесса, помеченные как доступные только для чтения. Это позволяет использовать механизм "копирования при записи" (copy-on-write), который экономит память
3. **Настройка регистров**: регистры дочернего процесса настраиваются, и процесс готов к запуску
4. **Запуск процесса**: дочерний процесс вызывает системный вызов exec, который загружает исполняемый файл и подготавливает новый адресное пространство для выполнения программы.

Реализация потоков в UNIX

Потоки в UNIX могут поддерживаться ядром или реализовываться полностью на уровне пользовательского пространства. В системах, где ядро поддерживает многозадачность с потоками, процесс с несколькими потоками при вызове fork сталкивается с определёнными сложностями.

Проблемы реализации потоков

1. **Поведение при fork**
   - Если процесс с несколькими потоками вызывает fork, возникает вопрос: должны ли все потоки быть скопированы в дочерний процесс? Это сложный вопрос, потому что, с одной стороны, копирование всех потоков может привести к проблемам синхронизации, например, когда один из потоков блокирован, ожидая ввода с клавиатуры, и другие потоки могут не получить нужный доступ к данным
2. **Синхронизация потоков**
   - Если поток в дочернем процессе удерживает мьютекс, а новый поток не может его освободить, то новый поток будет "висеть", ожидая освобождения мьютекса. Это серьёзная проблема, поскольку приводит к блокировкам и некорректному поведению
3. **Файловый ввод-вывод**
   - Когда один поток блокируется на операции ввода-вывода (например, чтение из файла), а другой поток пытается изменить состояние файла (например, закрыть его или переместить указатель), возникает неопределенность. Системе необходимо обеспечить правильную обработку таких ситуаций

7. Изложите концепции постраничной подкачки в различных системах UNIX.

Постраничная подкачка — это механизм управления памятью, при котором процессу не требуется целиком находиться в оперативной памяти. UNIX-системы эволюционировали от использования свопинга (полной выгрузки процессов на диск) к более гибкой подкачке страниц по требованию.

Системы UNIX на базе PDP-11, Interdata и начальные версии для VAX использовали свопинг — процессы полностью выгружались на диск и загружались обратно. С версии 3BSD была добавлена поддержка постраничной подкачки, позволившая эффективно работать с крупными программами.

Механизм подкачки:

1. Память делится на три части:
   - ядро операционной системы
   - карта памяти
   - оставшаяся память, разбитая на страничные блоки (используется для текста, данных, стека или свободных страниц)
2. Карта памяти содержит:

- информацию о содержимом каждой страницы
- ссылку на процесс, которому принадлежит страница
- тип сегмента (текст, данные, стек)
- смещение в сегменте
- флаги, нужные для алгоритма подкачки
- фиксированное место на диске, куда выгружается страница при нехватке памяти

3. Механизм подкачки реализуется двумя сущностями:
   - ядро, обрабатывающее страничные прерывания
   - страничный демон (процесс 2), который периодически запускается и следит за состоянием памяти

Если во время выполнения процесса происходит страничное прерывание (то есть процесс обращается к отсутствующей в памяти странице), ядро:

- ищет свободную страницу в списке;
- удаляет её из списка;
- загружает в неё требуемые данные с диска.

Если свободных страниц нет, выполнение процесса приостанавливается до тех пор, пока страничный демон не освободит память. Он очищает или выгружает неиспользуемые страницы, освобождая блоки для новых подкачек.

8. Раскройте особенности управления памятью в Linux.

Каждая программа в 32-разрядной Linux-системе получает 3 Гбайта виртуального адресного пространства, доступного только в пользовательском режиме, тогда как оставшийся 1 Гбайт зарезервирован для ядра и становится доступным, когда процесс переключается в режим ядра. Всё адресное пространство создаётся при запуске процесса и перезаписывается при системном вызове exec.

Виртуальное пространство процесса делится на однородные области, каждая из которых содержит страницы с одинаковыми правами доступа и режимами подкачки. Примеры таких областей — это сегменты кода, данных, стека и отображённые файлы. Каждая область описывается в структуре vm_area_struct, которая хранит параметры защиты, направление роста области, статус выгрузки и другие атрибуты. Эти структуры объединены в связный список, упорядоченный по виртуальным адресам. Если список становится слишком большим, он преобразуется в дерево для ускорения поиска.

Кроме того, Linux реализует механизм копирования при записи (copy-on-write): при вызове fork дочерний и родительский процессы получают ссылки на одни и те же страницы памяти, доступные только для чтения. Если один из процессов пытается произвести запись, создаётся отдельная копия страницы.

Некоторые области, например текстовые сегменты, используют двоичные файлы как резервное хранилище. Для других областей, таких как стек, резервное хранилище не назначается до тех пор, пока в этом не возникнет необходимость.

Трёхуровневая схема страничной подкачки

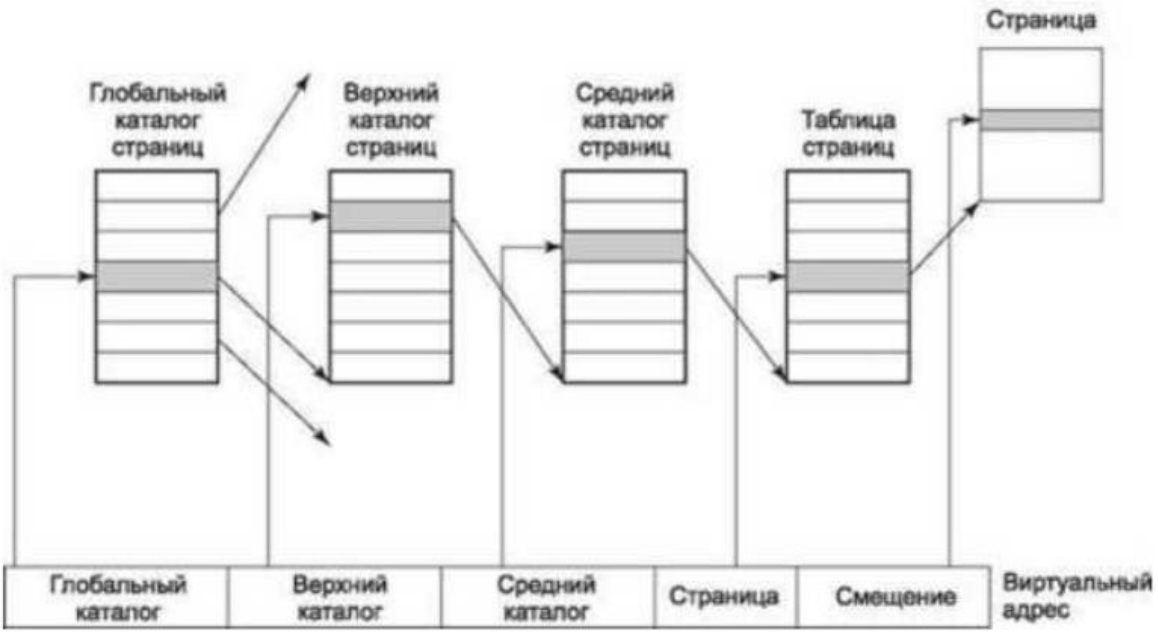

Рис. 3. Трехуровневая схема страничной подкачки

Система Linux использует трёхуровневую архитектуру страничной подкачки, которая особенно хорошо подходит для архитектур с большим адресным пространством, таких как Alpha. Эта схема представлена на рисунке, который вы приложили.

1. **Глобальный каталог страниц (Global Page Directory)** — это структура, содержащая указатели на каталоги верхнего уровня для каждого процесса. Индекс для доступа в этот каталог берётся из старших битов виртуального адреса.
2. **Верхний каталог страниц (Upper Directory)** — указывает на **средний каталог страниц**.
3. **Средний каталог страниц (Middle Directory)** — указывает на таблицу страниц.
4. **Таблица страниц (Page Table)** — содержит указатели на конкретные физические страницы памяти.
5. **Смещение (Offset)** — младшие биты виртуального адреса определяют смещение внутри найденной страницы.

Такой механизм позволяет поэтапно переходить от виртуального адреса к физическому, используя иерархию таблиц. При этом каждая таблица индексируется определённой частью виртуального адреса. На архитектурах, таких как Pentium, используется упрощённый двухуровневый вариант, в котором средний каталог содержит только одну запись.

Использование физической памяти

Физическая память в Linux делится между ядром, пользовательскими страницами, буферным кэшем файловой системы, страничным кэшем и другими служебными данными. Ядро загружается в фиксированную часть памяти и не выгружается. Буферный кэш содержит недавно использованные блоки файлов и конкурирует за память с пользовательскими страницами. Страничный кэш — это временное хранилище страниц, ожидающих выгрузки. Если такая страница снова потребуется, она может быть быстро возвращена в рабочее состояние.

Алгоритмы выделения памяти

Linux использует три основных алгоритма управления физической памятью:

1. «Дружественный» (buddy) алгоритм, при котором память делится на блоки размеров, кратных степеням двойки (1, 2, 4, 8 и т. д.). Это позволяет быстро находить подходящие блоки, но приводит к внутренней фрагментации.
2. Нарезка блоков, полученных от buddy-алгоритма, на более мелкие фрагменты. Это

помогает снизить фрагментацию.
3. Алгоритм для выделения непрерывной памяти в виртуальном адресном пространстве, когда физическая непрерывность не требуется.

9. Раскройте значение термина свопинг для UNIX. Сформулируйте преимущества и недостатки при его реализации.

Свопинг (swapping) в ранних версиях операционной системы UNIX (для компьютеров PDP-11, Interdata и начальной версии VAX) — это механизм перемещения данных между памятью и диском, управляемый верхним уровнем двухуровневого планировщика, называемым свопером (swapper). Основная цель свопинга заключалась в управлении ограниченным объемом свободной памяти ядра.

Выгрузка данных из памяти на диск (инициируемая свопером) происходила, когда у ядра заканчивалась свободная память по одной из следующих причин:
1. Системному вызову `fork` требовалась память для дочернего процесса.
2. Системный вызов `brk` собирался расширить сегмент данных.
3. Разросшемуся стеку требовалась дополнительная память.

Кроме того, свопинг использовался для освобождения места в памяти, когда требовалось запустить процесс, уже достаточно долго находящийся на диске.

Преимущества и недостатки при реализации свопинга:

Преимущества:
- **Возможность запуска больших программ:** Свопинг позволял запускать программы, размер которых превышал доступный объем физической памяти, за счет выгрузки неактивных частей программ на диск.
- **Управление дефицитом памяти:** Механизм свопинга давал возможность ядру реагировать на нехватку свободной памяти, освобождая ее путем перемещения данных на диск.
- **Гибкость в управлении процессами:** Свопер мог выбирать "жертву" для выгрузки из памяти, отдавая предпочтение блокированным процессам, чтобы не прерывать работу активных.

Недостатки:
- **Низкая производительность при активном свопинге:** "Чтобы не терять большую часть производительности системы на свопинг, ни один процесс не выгружался на диск, если он пробыл в памяти менее 2 с." Это указывает на то, что частый свопинг негативно сказывался на производительности.
- **Отсутствие постраничной подкачки (в ранних версиях):** Изначальный свопинг перемещал целые процессы или их сегменты между памятью и диском, а не отдельные страницы. Это означало, что для работы процессу требовалось быть целиком в памяти (до появления постраничной подкачки в 3BSD).
- **Выгрузка целых процессов:** Когда ядру не хватало памяти или нужно было загрузить процесс с диска, часто требовалось удалить из памяти **другой процесс** целиком, что могло быть неэффективно, если требовалась лишь небольшая часть памяти.
- **Потенциальная неэффективность выбора "жертвы" для выгрузки:** Хотя свопер и отдавал предпочтение блокированным процессам, выбор основывался на "сумме приоритета и времени пребывания в памяти", что не всегда гарантировало оптимальный результат для общей производительности.
- **Сложность определения "легкого" и "тяжелого" свопинга:** Различие между легким и тяжелым свопингом (требующим дополнительного высвобождения памяти) подчеркивает, что процесс управления памятью был не всегда тривиальным и мог

приводить к дополнительным задержкам.

10. Изложите концепцию файлов и связей, возможностей блокировок в файловой системе UNIX.

Файл в системе UNIX — это последовательность байтов произвольной длины, которая может содержать любую информацию. Система не делает различий между текстовыми (ASCII) файлами, двоичными файлами или любыми другими типами. Значение битов в файле полностью определяется его владельцем; системе это безразлично. Изначально имена файлов были ограничены 14 символами, но в Berkeley UNIX этот предел был расширен до 255 символов, что стало стандартом. В именах файлов разрешены все ASCII-символы, кроме NUL.
Существует два способа задания имени файла:

- **Абсолютный путь**: Указывает, как найти файл от корневого каталога (например, `/usr/ast/books/mos2/chap-10`).
- **Относительный путь**: Указывается относительно рабочего (текущего) каталога. Например, если `/usr/ast/books/mos2` является рабочим каталогом, то `chap-10` является относительным путём к файлу `chap-10` в этом каталоге.

Помимо обычных файлов, UNIX поддерживает:

- **Символьные специальные файлы**: Используются для моделирования последовательных устройств ввода-вывода (например, клавиатуры `/dev/tty` и принтеры `/dev/lp`).
- **Блочные специальные файлы**: Могут использоваться для чтения и записи необработанных дисковых разделов, минуя файловую систему (например, `/dev/hd1`). Они используются для страничной подкачки, свопинга и программами для работы с файловой системой (например, `mkfs`, `fsck`).

Связи

Связи в UNIX решают проблему неудобства использования длинных абсолютных путей к файлам, принадлежащим другим пользователям или расположенным в других местах файлового дерева. Связи представляют собой записи каталога, которые указывают на другие файлы, позволяя обращаться к ним по более коротким или удобным именам.

Блокировки

Блокировки в файловой системе UNIX позволяют избежать конфликтов при одновременном использовании одного и того же файла двумя и более процессами. Это особенно важно в приложениях, таких как базы данных, где независимые пользователи могут обращаться к одним и тем же файлам.
Стандарт определяет два типа блокировки:

- **Блокировка без монополизации (shared lock)**: Если часть файла уже содержит блокировку без монополизации, установка повторной такой блокировки на это место разрешается. Однако попытка установки блокировки с монополизацией будет отвергнута.
- **Блокировка с монополизацией (exclusive lock)**: Если какая-либо область файла содержит блокировку с монополизацией, любые попытки заблокировать любую часть этой области будут отвергаться до тех пор, пока монопольная блокировка не будет снята. Для успешной установки такой блокировки каждый байт в данной области должен быть доступен.

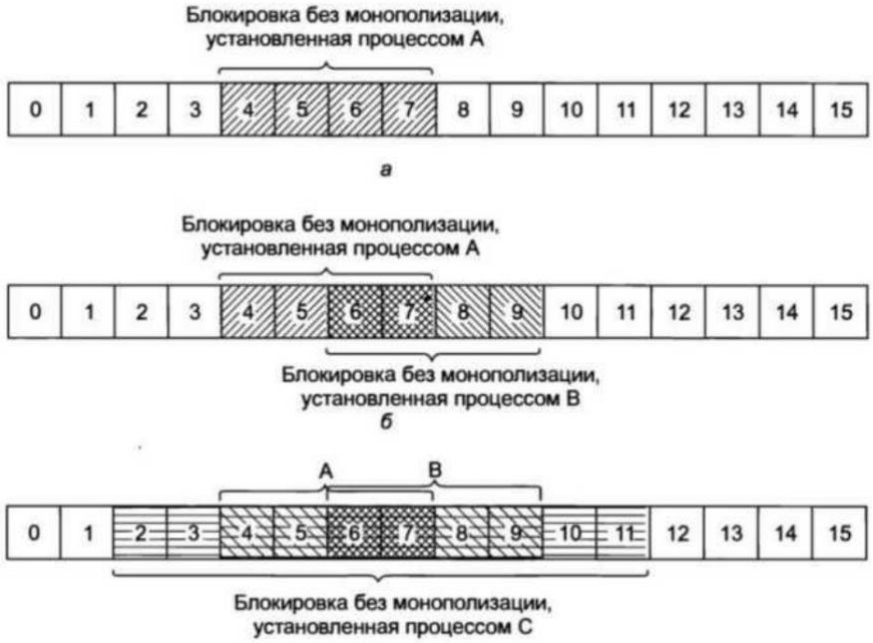

Рис. 4. Пример перекрытия заблокированных областей

При установке блокировки процесс может указать, хочет ли он получить управление немедленно или будет ждать, пока блокировка не будет установлена:

- **Вызов с ожиданием**: Процесс блокируется до тех пор, пока с запрашиваемой области файла не будет снята блокировка, установленная другим процессом. После этого процесс активизируется, и ему сообщается об установке блокировки.
- **Вызов без ожидания**: Процесс немедленно получает ответ об успехе или неудаче операции.

Заблокированные области могут перекрываться. Например, несколько процессов могут установить блокировки без монополизации на перекрывающиеся области файла одновременно. Если же процесс пытается установить блокировку с монополизацией на область, которая уже имеет какие-либо блокировки (даже без монополизации, если они перекрываются), он будет заблокирован до снятия всех конфликтующих блокировок.

11. Раскройте сущность системных вызовов для работы с файлами и каталогами.

Системные вызовы являются интерфейсом между пользовательскими программами и ядром операционной системы, предоставляя средства для взаимодействия с файловой системой. В UNIX существует ряд системных вызовов, предназначенных для манипулирования файлами и каталогами.

Системные вызовы для работы с отдельными файлами:

1. **`creat` (или `create`):**
   - **Назначение:** Создает новый файл. Если файл с таким именем уже существует, его длина усекается до нуля, и все содержимое теряется.
   - **Параметры:** Принимает имя файла и режим защиты (`mode`). Режим защиты определяет права доступа к файлу для различных категорий пользователей (владелец, группа, остальные).
   - **Возвращаемое значение:** В случае успеха возвращает небольшое неотрицательное целое число, называемое **дескриптором файла (file descriptor, fd)**. Этот дескриптор используется в последующих системных вызовах для обращения к файлу.

- **Дополнительно:** `creat` не только создает файл, но и открывает его для записи.

2. **`read`:**
   - **Назначение:** Читает данные из открытого файла.
   - **Параметры:** Дескриптор файла (`fd`), буфер для хранения прочитанных данных (`buffer`) и количество байтов для чтения (`nbytes`).
   - **Механизм:** С каждым открытым файлом связан указатель на текущую позицию. При последовательном чтении этот указатель автоматически перемещается на следующий байт после последнего прочитанного.
3. **`write`:**
   - **Назначение:** Записывает данные в открытый файл.
   - **Механизм:** Подобно `read`, `write` также использует указатель текущей позиции в файле и перемещает его после записи.
4. **`lseek`:**
   - **Назначение:** Перемещает указатель текущей позиции в файле, позволяя осуществлять доступ к произвольной части файла (случайный доступ) или даже за его концом.
   - **Параметры:** Дескриптор файла (`fd`), новая позиция в файле (смещение) и третий параметр, указывающий, относительно чего задана эта позиция (начала файла, конца файла или текущей позиции).
   - **Возвращаемое значение:** Абсолютная позиция в файле после перемещения указателя.
   - **Важный момент:** `lseek` не вызывает физического перемещения головок диска; он просто обновляет числовую позицию в памяти, связанную с дескриптором файла.

Системные вызовы для работы с каталогами:

В классической системе UNIX программы могли напрямую читать содержимое каталогов, так как они представляли собой простой набор 16-байтовых записей. Однако с введением длинных имен файлов в Berkeley UNIX и изменением внутренней структуры каталогов, возникла необходимость в абстрагировании от деталей реализации каталогов. Для этого были разработаны следующие системные вызовы:

1. **`opendir`:**
   - **Назначение:** Открывает каталог для чтения.
   - **Возвращаемое значение:** Возвращает указатель на структуру, представляющую открытый каталог.
2. **`readdir`:**
   - **Назначение:** Читает следующую запись из открытого каталога.
   - **Возвращаемое значение:** Возвращает указатель на структуру, содержащую информацию об одной записи каталога (например, имя файла и номер i-узла).
3. **`closedir`:**
   - **Назначение:** Закрывает открытый каталог, освобождая связанные с ним ресурсы.
4. **`rewinddir`:**
   - **Назначение:** Перемещает указатель чтения каталога в начало, позволяя повторно перебирать записи.

Эти системные вызовы для работы с каталогами обеспечивают переносимость программ, так как они не зависят от внутренней структуры каталогов, которая может меняться между версиями UNIX. Впоследствии они были приняты во всех других версиях UNIX и стандарте POSIX.

Концепция дескриптора файла и его роль:

Успешный системный вызов `creat` возвращает **дескриптор файла**. Это небольшое целое число, которое является индексом в таблице дескрипторов файла, специфичной для

каждого процесса. Каждая запись в этой таблице содержит информацию об открытом файле, включая:

- **Указатель на i-узел:** Несмотря на кажущуюся простоту, прямой указатель на i-узел в таблице дескрипторов файла не всегда подходит.
- **Указатель на текущую позицию в файле:** Это критически важный элемент. Каждый процесс, открывающий файл, должен иметь свой собственный указатель текущей позиции. Если бы этот указатель хранился в i-узле (разделяемом для всех процессов, открывших файл), то при совместном использовании файла разными процессами (например, в сценарии оболочки `p1 > x` и `p2 > x`) возникли бы проблемы с корректным позиционированием для записи. Поэтому указатель на текущую позицию **хранится в таблице дескрипторов файла** (или в структуре, на которую указывает запись в этой таблице), обеспечивая независимую позицию для каждого процесса, открывшего файл.

12. Опишите реализацию традиционной файловой системы UNIX.

Реализация файловой системы UNIX

В традиционной файловой системе UNIX дисковый раздел содержит файловую систему со следующей структурой:

- **Блок 0**: Не используется системой, часто содержит программу загрузки.
- **Блок 1 (суперблок)**: Хранит критическую информацию о файловой системе, включая количество i-узлов, количество дисковых блоков и начало списка свободных блоков. Повреждение суперблока делает файловую систему нечитаемой.
- **I-узлы (индексные узлы)**: Располагаются после суперблока. Каждый i-узел имеет длину 64 байта и описывает ровно один файл, содержая учетную информацию (возвращаемую `stat`) и достаточно данных для нахождения всех блоков файла на диске.
- **Блоки данных**: Располагаются после i-узлов и хранят все файлы и каталоги. Блоки большого файла могут быть разбросаны по всему диску.

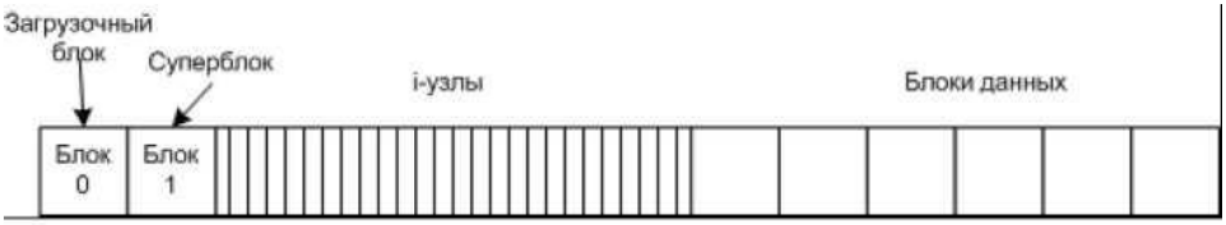

Рис. 5. Структура файловой системы дискового раздела в UNIX

Структура каталога

В традиционной файловой системе UNIX (V7) каталог сам по себе является файлом. Он представляет собой:

- Несортированный набор 16-байтовых записей**.**
- Каждая запись в каталоге состоит из двух основных полей:
  - 14-байтного имени файла**.**
  - Номера i-узла (2 байта, так как i-узлов не могло быть больше 65536).
- Когда система пытается открыть файл в рабочем каталоге, она линейно считывает каталог, сравнивая искомое имя файла с каждой записью, пока не найдет совпадение или не достигнет конца каталога.
- Если файл найден, система извлекает его номер i-узла и использует его как индекс в таблице i-узлов на диске, чтобы найти соответствующий i-узел и считать его в память. Этот i-узел затем помещается в таблицу i-узлов в ядре**,** которая содержит i-узлы всех открытых в данный момент файлов и каталогов.

Как происходит чтение файла

Когда пользовательская программа вызывает системный вызов `read(fd, buffer, nbytes)`:

1. Ядро получает эти три параметра.
2. Используя `fd` (дескриптор файла), ядро обращается к массиву дескрипторов файла (который является внутренней таблицей для каждого процесса).
3. Запись в этом массиве, соответствующая `fd`, содержит указатель на текущую позицию в файле**.** Важно, что этот указатель хранится именно в таблице дескрипторов файла, а не в i-узле. Это позволяет нескольким независимым процессам одновременно открывать один и тот же файл, при этом каждый процесс будет иметь свою собственную уникальную позицию чтения/записи, не мешая другим.
4. Через дескриптор файла файловая система находит соответствующий i-узел файла. I-узел предоставляет информацию о расположении блоков данных файла на диске.
5. Используя текущую позицию и информацию из i-узла, ядро определяет, какой блок данных нужно прочитать, и осуществляет операцию ввода-вывода с диском.
6. Прочитанные данные помещаются в предоставленный буфер (`buffer`).
7. Указатель текущей позиции в файле обновляется, указывая на следующий байт после прочитанных данных.

13. Опишите реализацию файловой системы Berkeley Fast.

Файловая система Berkeley Fast File System (FFS) была разработана для устранения недостатков традиционной файловой системы UNIX (Unix File System, UFS), таких как низкая производительность из-за фрагментации и ограниченная длина имен файлов. FFS внесла ряд существенных улучшений, направленных на повышение эффективности и функциональности.

Вот основные особенности реализации файловой системы Berkeley Fast:

1. **Реорганизация каталогов и поддержка длинных имен файлов:**
   - В отличие от 14-символьного ограничения традиционного UNIX, FFS увеличила длину имен файлов до 255 символов.
   - Это изменение повлекло за собой необходимость переработки внутренней структуры каталогов. Каталоги больше не являются простыми несортированными наборами фиксированных 16-байтовых записей.
   - **Структура каталоговой записи в BSD:** Каждая запись в каталоге состоит из:
     - **Номера i-узла:** Указывает на соответствующий i-узел файла или каталога.
     - **Размера всей каталоговой записи в байтах:** Это поле необходимо для быстрого перехода к следующей записи, так как записи имеют переменную длину. Могут присутствовать дополнительные байты-заполнители в конце записи.
     - **Поля типа файла:** Определяет, является ли запись обычным файлом, каталогом и т.д.
     - **Длины имени файла в байтах:** Указывает фактическую длину имени.
     - **Само имя файла:** Имеет переменную длину, заканчивается нулевым байтом и может быть дополнено до 32-байтовой границы. За ним могут следовать дополнительные байты-заполнители.
   - **Обработка удаления файлов:** При удалении файла в каталоге FFS не удаляет запись физически, а увеличивает размер записи предыдущего файла, превращая байты удаленной записи в заполнители. Эти байты могут быть повторно использованы при создании нового файла.
   - **Системные вызовы для работы с каталогами:** Поскольку внутренняя структура каталогов стала сложнее и переменной, для обеспечения совместимости и удобства были введены специальные системные вызовы для работы с каталогами: `opendir`,

`closedir`, `readdir` и `rewinddir`. Эти вызовы позволяют программам читать каталоги, не зная их внутренней структуры, что делает их более переносимыми.

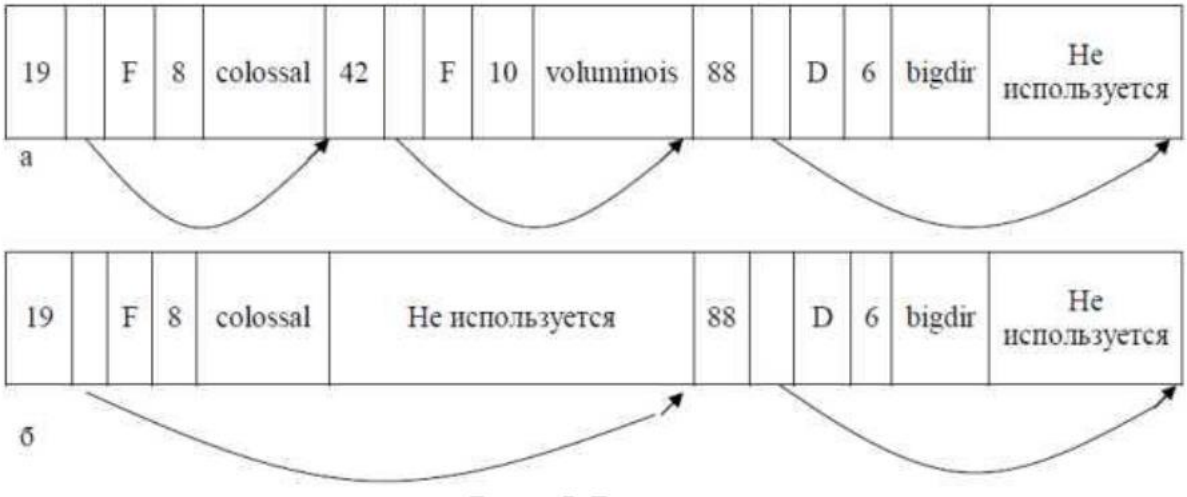

Рис. 6. Структура каталоговой записи в файловой системе Berkeley Fast

2. **Группы цилиндров (Cylinder Groups):**
   - Одно из самых значительных улучшений FFS – это разбиение диска на группы цилиндров. Каждая группа цилиндров имеет свою собственную структуру, аналогичную мини-файловой системе:
   - Копию суперблока (для повышения отказоустойчивости и ускорения доступа к метаданным).
     - Собственные **i-узлы**.
     - Собственные **блоки данных**.
     - Битовые карты свободных блоков и i-узлов для быстрой аллокации.
   - **Цель:** Основная идея заключается в повышении локальности размещения данных. При обращении к файлам снижается время, затрачиваемое жестким диском на перемещение блоков головок, поскольку i-узел файла и его блоки данных (по возможности) хранятся в одной и той же группе цилиндров или в ближайших группах. Это уменьшает количество и дальность перемещений головок диска (seek times), значительно улучшая производительность.
   - **Политика размещения:** FFS стремится размещать i-узел и блоки данных одного файла в одной группе цилиндров. Если файл растет и ему требуются новые блоки, система старается выделить их в той же группе цилиндров, где находится i-узел, или в соседних.
3. **Использование блоков двух размеров:**

   Традиционные файловые системы использовали блоки фиксированного размера, что приводило либо к неэффективному использованию дискового пространства (для маленьких файлов, когда большая часть блока оставалась неиспользованной), либо к неэффективному чтению/записи (для больших файлов, требовавших большого количества мелких блоков).

   FFS ввела два размера блоков:
   - **Крупные блоки:** Используются для больших файлов. Это позволяет хранить большие объемы данных в меньшем количестве блоков, уменьшая количество указателей в i-узле и повышая скорость последовательного чтения/записи.
   - **Мелкие блоки (фрагменты):** Используются для небольших файлов или для последней, частично заполненной части крупного блока. Это значительно сокращает потери дискового пространства из-за внутренней фрагментации для маленьких файлов.

     Компромисс: Наличие блоков двух размеров обеспечивает эффективное

использование дискового пространства для небольших файлов и эффективное чтение/запись для больших файлов, но увеличивает сложность реализации файловой системы.

4. **Кэширование имен (Name Caching):**

   Для ускорения поиска файлов в каталогах, особенно в больших каталогах, где линейный поиск может быть медленным, FFS добавила кэширование имен файлов.

   Перед тем как выполнять линейный поиск имени в каталоге, система проверяет кэш. Если имя файла найдено в кэше, нет необходимости обращаться к диску и выполнять медленный поиск в каталоге, что значительно повышает производительность.

14. Опишите реализацию файловой системы Ext2.

Файловая система Ext2 реализована на жестком диске путем деления его на группы блоков, независимо от границ цилиндров. Каждая группа блоков начинается с суперблока, который содержит информацию о количестве блоков и i-узлов в данной группе, а также о размере группы блоков. Затем следует описатель группы, содержащий информацию о расположении битовых массивов, количестве свободных блоков и i-узлов в группе, а также количестве каталогов в группе.

Учет свободных блоков и свободных i-узлов ведется в двухбитовых массивах, размер каждого из которых равен одному блоку. Это ограничивает размер группы блоков 8192 блоками и 8192 i-узлами при размере блоков в 1 Кбайт.

После битовых массивов располагаются сами i-узлы. Размер каждого i-узла составляет 128 байт. В i-узле Ext2 используются 12 прямых и 3 косвенных дисковых адреса, а длина адресов увеличена до 4 байт, что позволяет поддерживать дисковые разделы размером более 16 Гбайт. Также зарезервированы поля для указателей на списки управления доступом.

Файловая система Ext2 использует дисковые блоки только одного размера — 1 Кбайт. При увеличении размера файла Ext2 пытается поместить новый блок файла в ту же группу блоков, что и остальные блоки, желательно сразу после предыдущих блоков. При создании нового файла в каталоге Ext2 старается выделить ему блоки в той же группе блоков, в которой располагается каталог. Новые каталоги, наоборот, равномерно распределяются по всему диску.

Рис. 7. Размещение файловой системы Ext2 на жестком диске

15. Опишите особенности Ext3 и Ext4.

Особенности Ext3:

- **Журналируемая файловая система:** Ext3 использует журнал для записи всех операций файловой системы в последовательном порядке. Это позволяет восстановить файловую систему после сбоев или отключения питания, просматривая журнал и выполняя незафиксированные изменения.
- **Совместимость с Ext2:** Ext3 в значительной степени совместима с Ext2. Основные

структуры данных и компоновка диска у них одинаковы. Размонтированная файловая система Ext2 может быть смонтирована как Ext3 с поддержкой журналирования.

- **Механизм журналирования JBD:** Для работы с журналом используется отдельное блочное устройство журналирования JBD (Journaling Block Device). JBD поддерживает три структуры данных: запись журнала (log record), описатель атомарной операции (atomic operation handle) и транзакцию (transaction).
- **Настройка журналирования:** Ext3 можно настроить на журналирование всех изменений на диске или только изменений, относящихся к метаданным файловой системы (i-узлы, суперблоки, битовые массивы и т. д.). Журналирование только метаданных снижает издержки и повышает производительность, но не гарантирует целостность данных в файлах.
- **Журнал как кольцевой буфер:** Журнал является файлом, работающим как кольцевой буфер. Он может храниться как на том же устройстве, что и основная файловая система, так и на другом.

Особенности Ext4:

- **Увеличенный максимальный объем раздела:** Ext4 увеличивает максимальный объем раздела до одного экзабайта.
- **Прирост производительности:** За счет реализации механизма пространственной записи файлов (новая информация добавляется в конец заранее выделенной по соседству области файла) Ext4 демонстрирует прирост производительности.
- **Снижение фрагментации:** Механизм пространственной записи файлов приводит к значительному снижению фрагментации.

16. Опишите архитектуру файловой системы NFS.

Архитектура файловой системы NFS (Network File System) строится вокруг концепции совместного использования файловой системы произвольным количеством клиентов и серверов.

Основные элементы архитектуры:

- **Клиенты и Серверы:** NFS позволяет любому компьютеру выступать как клиентом (потребляющим данные), так и сервером (предоставляющим данные). Хотя системы могут находиться в глобальной сети, чаще всего они функционируют в пределах одной локальной сети.
- **Экспорт Каталогов (Серверная сторона):** Сервер NFS отвечает за предоставление удаленного доступа к своим каталогам. Он "экспортирует" один или несколько каталогов, делая их доступными для клиентов. Как правило, при экспорте каталога становятся доступными и все его подкаталоги, то есть экспортируется целое дерево каталогов. Информация об экспортируемых каталогах хранится в файле `/etc/exports`, что обеспечивает их автоматический экспорт при запуске сервера.
- **Монтирование Каталогов (Клиентская сторона):** Клиенты получают доступ к экспортированным каталогам путем их "монтирования". Когда клиент монтирует удаленный каталог, этот каталог становится частью его собственной локальной иерархии файловой системы, интегрируясь в существующее дерево каталогов клиента. Важно отметить, что клиент сам определяет, в каком месте своей иерархии он монтирует удаленный каталог; сервер не имеет информации о том, куда клиент монтирует его каталоги.

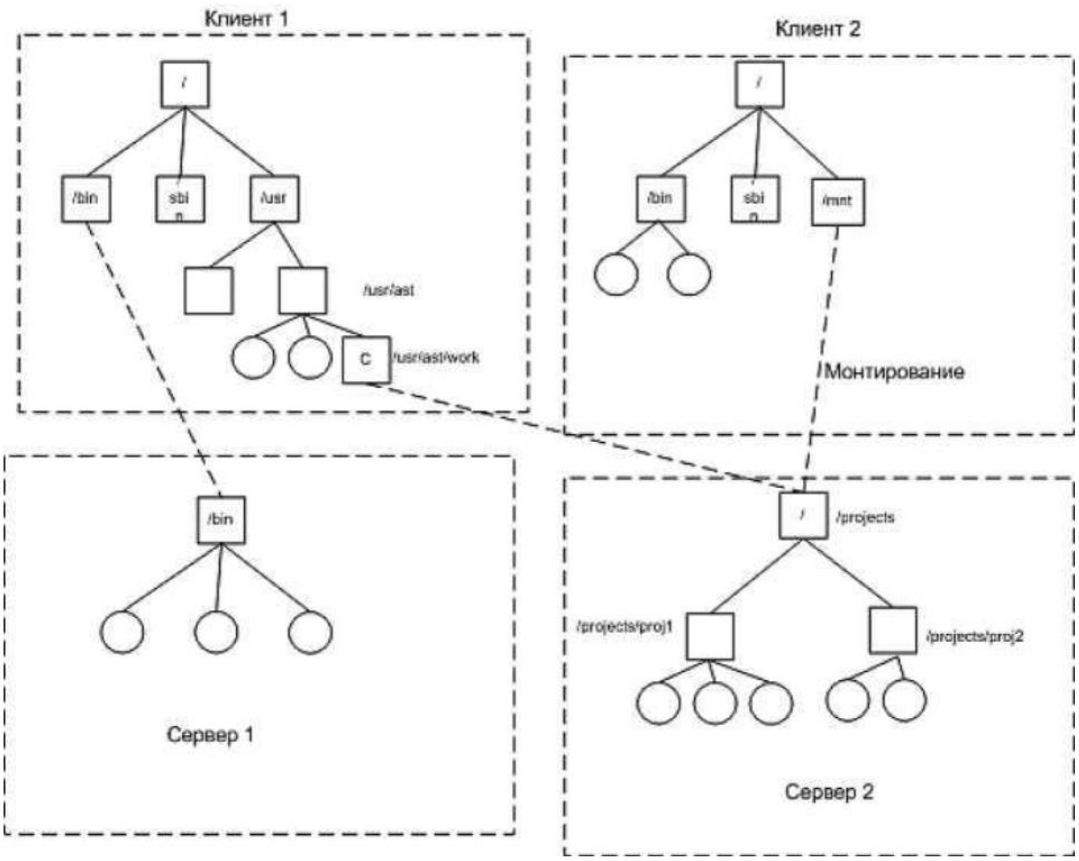

Рис. 8. Пример монтирования удаленного каталога (Каталоги показаны на рисунке в виде квадратов, а файлы — в виде кружков.)

17. Опишите протоколы файловой системы NFS

Протоколы файловой системы NFS разработаны для обеспечения бесшовного взаимодействия между разнообразными системами, позволяя клиентам и серверам, работающим под разными операционными системами и на различном оборудовании, обмениваться данными. Для этого используются два ключевых протокола, каждый из которых выполняет свою специфическую функцию.

Протокол монтирования каталогов

Этот протокол отвечает за первоначальное установление связи между клиентом и сервером, позволяя клиенту присоединить удаленный каталог к своей локальной файловой иерархии. Когда клиент хочет смонтировать каталог, он отправляет серверу запрос, указывая путь к желаемому каталогу. При этом серверу не сообщается, куда именно клиент планирует смонтировать этот каталог на своей стороне, поскольку для сервера эта информация не имеет значения. Если указанный путь верен и каталог был экспортирован сервером, сервер в ответ отправляет клиенту специальный **дескриптор файла**. Этот дескриптор содержит всю необходимую информацию для однозначной идентификации каталога, включая тип файловой системы, диск, i-узел каталога и сведения о правах доступа. Именно этот дескриптор будет использоваться во всех последующих операциях чтения и записи файлов в смонтированном каталоге и его подкаталогах.

Существуют два основных подхода к монтированию удаленных файловых систем:

- **Статическое монтирование** предполагает, что команды монтирования прописываются в стартовом сценарии UNIX-системы, например, в `/etc/rc`. Это гарантирует, что все необходимые удаленные файловые системы будут автоматически смонтированы еще до того, как пользователи смогут войти в систему. Однако, у этого метода есть существенный недостаток: если один из серверов, перечисленных в сценарии,

недоступен, это может создать трудности при запуске клиента, вызвать задержки и ошибки, даже если пользователю в данный момент этот сервер не нужен.

- **Автомонтирование** предлагает более гибкий подход. При этом методе локальный каталог может быть ассоциирован с несколькими удаленными каталогами, но ни один из них не монтируется при загрузке операционной системы. Монтирование происходит только по требованию: когда клиент впервые обращается к файлу в таком ассоциированном каталоге, операционная система отправляет запросы всем соответствующим серверам. Каталог того сервера, который ответит первым, будет смонтирован. Это значительно повышает устойчивость системы к сбоям, так как работоспособность клиента не зависит от доступности всех серверов, достаточно, чтобы хотя бы один из них ответил. Кроме того, это может улучшить производительность, поскольку первый ответивший сервер, скорее всего, является наименее загруженным.

Протокол доступа к каталогам и файлам

После того как каталог смонтирован, в дело вступает второй протокол NFS, предназначенный для непосредственного управления файлами и каталогами. Он позволяет клиентам отправлять серверу различные команды, такие как создание, удаление, чтение и запись файлов. Клиенты также могут получать доступ к атрибутам файлов, включая их режим, размер и время последнего изменения. NFS поддерживает большинство стандартных системных вызовов UNIX, за исключением, что примечательно, системных вызовов `open` и `close`.

Что касается безопасности, NFS изначально использовала стандартные UNIX-права доступа (биты `rwx` для владельца, группы и других). Каждый запрос просто содержал идентификаторы пользователя и группы вызывающего процесса, которые сервер использовал для проверки прав. Однако, опыт показал, что это предположение о доверии клиентов было слишком наивным. В современных реализациях для обеспечения надежной аутентификации клиента и сервера при каждом запросе и ответе используется шифрование с открытым ключом. Это значительно повышает безопасность, предотвращая возможность злоумышленника выдать себя за другого клиента или сервер, поскольку ему неизвестен секретный ключ.

18. Изложите концепцию реализации файловой системы NFS.

В большинстве систем UNIX используется трехуровневая реализация, сходная с изображенной на рисунке

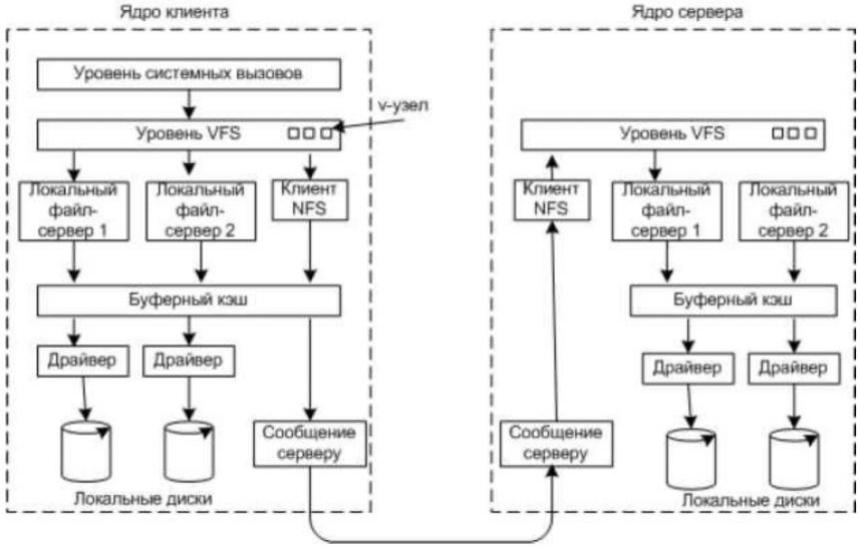

Рис. 9. Трехуровневая реализация файловой системы NFS

Уровень системных вызовов – верхний уровень, отвечает за управление системными вызовами, например open, read, close.

Уровень VFS(Virtual File System) – второй уровень, отвечает за управление таблицей, содержащей по одной записи для каждого открытого файла, аналогичной таблице i- узлов для открытых файлов в системе UNIX. Уровень VFS содержит для каждого открытого файла записи, называемые v-узлами (virtual i-node — виртуальный i-узел). V-узлы используются, чтобы отличать локальные файлы от удаленных. Для удаленных файлов предоставляется информация, достаточная для доступа к ним.

Пример работы:

1. Для монтирования удалённой файловой системы программа mount:
   a. Анализирует имя удаленного каталога и обнаруживает имя сервера NFS
   b. У сервера запрашивается дескриптор удаленного каталога
   c. Если дескриптор пришел, то идет обращение к системному вызову mount
2. При последующих системных вызовах:
   a. Уровень VPS находит соответствующий V-узел, по которому определяет, локальный ли файл или удалённый
   b. Серверу посылается дескриптор, смещение в файле и количество байтов

Для повышения эффективности:

1. Обмен информацией осуществляется большими порциями по 8192 байт
2. Кеширование на серверах
3. Кеширование на клиенте (i-узлы и данные)

19. Перечислите основные системные вызовы, используемые при работе с файлами и каталогами в UNIX. Опишите поля структуры, возвращаемой системным вызовом stat.

Таблица 1

Основные системные вызовы для работы с файлами:

Системный вызов	**Назначение**
`open()`	Открытие файла (может создавать файл, если задан флаг `O_CREAT`)
`read()`	Чтение данных из файла
`write()`	Запись данных в файл
`close()`	Закрытие открытого дескриптора файла
`lseek()`	Смещение позиции чтения/записи в файле
`stat()`	Получение информации о файле
`fstat()`	То же самое, но по файловому дескриптору
`lstat()`	Как `stat()`, но не разыменовывает символьные ссылки
`unlink()`	Удаление файла
`rename()`	Переименование файла или каталога
`chmod()`	Изменение прав доступа
`chown()`	Изменение владельца/группы
`truncate()`	Изменение размера файла
`fsync()`	Сброс данных на диск
`access()`	Проверка прав доступа к файлу

Таблица 2

Основные вызовы для работы с каталогами:

Системный вызов	**Назначение**
`mkdir()`	Создание нового каталога
`rmdir()`	Удаление пустого каталога
`opendir()` / `readdir()` / `closedir()`	Работа с содержимым каталога (через `DIR*`)
`chdir()`	Изменение текущего рабочего каталога
`getcwd()`	Получение текущего рабочего каталога

Структура `stat` (обычно `struct stat`). Системный вызов `stat()` возвращает информацию о файле в структуре `struct stat`, определённой в `<sys/stat.h>`:

```
struct stat {
   dev_t     st_dev;     // ID устройства, на котором находится файл
   ino_t     st_ino;     // Иномер (уникальный идентификатор) файла
   mode_t    st_mode;    // Тип файла и права доступа
   nlink_t   st_nlink;   // Количество жёстких ссылок
   uid_t     st_uid;     // UID владельца
   gid_t     st_gid;     // GID группы
   dev_t     st_rdev;    // Тип устройства (если специальный файл)
   off_t     st_size;    // Размер файла в байтах
   blksize_t st_blksize; // Размер блока ввода-вывода
   blkcnt_t  st_blocks;  // Количество выделенных блоков

   time_t    st_atime;   // Время последнего доступа
   time_t    st_mtime;   // Время последней модификации
   time_t    st_ctime;   // Время последнего изменения метаданных
};
```

20. Изложите концепцию безопасности, реализованную в UNIX.

У каждого пользователя в UNIX-системе есть уникальный UID, файлы имеют владельцев (изначально пользователь, создавший файл). Пользователь может принадлежать к группе/группам, у которой/которых есть уникальный GID. Процессы несут в себе UID и GID. Если процесс создаёт файл, то этот файл получает UID и GID процесса. разрешения определяют доступ к этому файлу для владельца файла, для других членов группы владельца файла и для всех прочих пользователей. Для каждой из этих трех категорий определяется три вида доступа: чтение, запись и исполнение файла, что обозначается соответственно буквами r,w,x (read, write, execute). Для каталогов X – это возможность поиска в нём. Пользователь с UID = 0 – суперпользователь, имеет доступ ко всем операциям над всеми файлами. Процессы с UID = 0 может вызывать небольшую группу системных вызовов, к которым не могут обращаться обычные пользователи. Для устройств работает тот же механизм rwx, потому что устройства в системе – это те же файлы.

Биты SETUID и SETGID позволяют изменять данные в файлах, принадлежащих суперпользователю (но только те, которые относятся к ним самим).

Системные вызовы безопасности:

- s=chmod(path, mode) Изменить режим защиты файла
- s=access(path, mode) Проверить разрешение доступа к файлу,
- используя действительные UID и GID
- uid=getuid() Получить действительный UID

- uid=geteuid() Получить рабочий UID
- gid=getgid Получить действительный GID
- gid=getegid() Получить рабочий GID
- s=chown(path, owner, group) Изменить владельца и группу
- s=setuid(uid) Установить UID
- s=setgid(gid) Установить GID

21. Раскройте сущность управления памятью в UNIX.

У каждого процесса в системе UNIX есть адресное пространство, состоящее из трех сегментов: текста, данных и стека.

Текстовый (программный) сегмент содержит машинные команды, образующие исполняемый код программы. Он создается компилятором и ассемблером при трансляции программы, написанной на языке высокого уровня (например, C или C++) в машинный код. Как правило, текстовый сегмент разрешен только для чтения.

Сегмент данных содержит переменные, строки, массивы и другие данные программы. Он состоит из двух частей: инициализированных данных и неинициализированных данных (BSS (Bulk Storage System— запоминающее устройство большой емкости, массовое ЗУ)).

Используется для хранения локальных переменных, адресов возврата, параметров функций. При запуске в стеке находятся параметры командной строки и переменные окружения. Расширяется в сторону уменьшения адресов (в отличие от кучи).

Большинством систем UNIX поддерживаются текстовые сегменты совместного использования. На некоторых компьютерах аппаратное обеспечение поддерживает раздельные адресные пространства для команд и для данных. Если такая возможность есть, система UNIX может ею воспользоваться.

Многими версиями UNIX поддерживается отображение файлов на адресное пространство памяти. Это свойство позволяет отображать файл на часть адресного пространства процесса, так чтобы можно было читать из файла и писать в файл, как если бы это был массив, хранящийся в памяти.

Дополнительное преимущество отображения файла на память заключается в том, что два или более процессов могут одновременно отобразить на свое адресное пространство один и тот же файл. Запись в этот файл одним из процессов мгновенно становится видимой всем остальным.

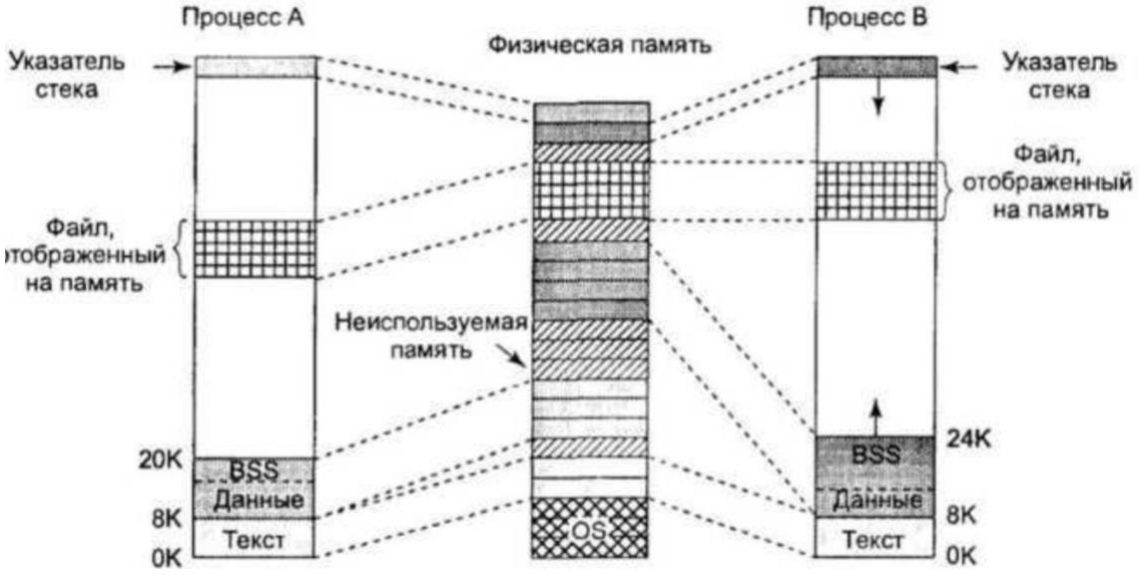

Рис. 10. Файл, одновременно отображенный на адресные пространства двух процессов по различным виртуальным адресам.

Свопинг в UNIX осуществлялся специальным компонентом ядра — свопером (swapper), который управлял перемещением процессов между оперативной памятью и диском. Свопинг инициировался, когда системе не хватало памяти, например, при вызове fork, расширении сегмента данных (brk) или росте стека. Чтобы освободить место, свопер выбирал процесс-кандидат на выгрузку, отдавая предпочтение заблокированным процессам, которые временно не могли выполняться. Из них выбирался процесс с наибольшим временем нахождения в памяти и наивысшим приоритетом.

Периодически свопер проверял, не готовы ли выгруженные процессы к возобновлению. Если готовых процессов было несколько, выбирался тот, что дольше всего находился на диске. Если процесс можно было загрузить без удаления других — происходил лёгкий свопинг. В противном случае, если требовалось выгрузить другие процессы — это называлось тяжёлым свопингом.

Процессы не выгружались, если они пробыли в памяти менее 2 секунд, чтобы избежать излишней потери производительности. Управление свободной памятью и пространством на диске осуществлялось через связные списки свободных участков — при необходимости выделялся первый подходящий блок, а остаток возвращался в список.

Постраничная подкачка появилась в UNIX начиная с версии 3BSD, чтобы поддерживать выполнение программ, превышающих по объёму доступную оперативную память. Суть её в том, что для запуска процесса вовсе не требуется загружать в память весь его код и данные — достаточно лишь таблиц страниц и структуры пользователя. Оставшиеся страницы (текста, данных, стека) загружаются **по мере обращения к ним**, что делает подкачку **по требованию**.

В Berkeley UNIX (4BSD) отсутствует модель рабочего набора и другие формы предсказательной подкачки, поскольку архитектура VAX не предоставляла аппаратной поддержки (например, битов обращения), а программная реализация была бы слишком ресурсоёмкой. Вместо этого подкачка реализуется совместно **ядром** и отдельным **страничным демоном** — специальным процессом, который периодически проверяет объём свободных страниц и при необходимости инициирует их освобождение.

В памяти UNIX-системы 4BSD три основные части: ядро, карта памяти и пользовательская память. Ядро и карта памяти находятся в памяти постоянно, а остальная часть разбивается на страничные блоки, каждый из которых может быть занят под текст, данные или стек, или находиться в списке свободных. **Карта памяти** содержит для каждой страницы информацию о её владельце, типе содержимого, смещении и различные флаги, необходимые алгоритму подкачки.

Когда процесс пытается обратиться к странице, которой нет в памяти, возникает **страничное прерывание**. Операционная система выделяет свободный страничный блок, считывает туда требуемые данные с диска и продолжает выполнение процесса. Если свободных страниц нет, выполнение процесса приостанавливается до тех пор, пока страничный демон не освободит нужный блок.

В UNIX с 4BSD замещение страниц управляется страничным демоном, который каждые 250 мс проверяет объём свободной памяти. Если он меньше значения `lotsfree` (обычно 1/4 ОЗУ), демон начинает выгружать страницы на диск, применяя модифицированный алгоритм часов. Он сбрасывает бит использования при первом проходе и удаляет страницу при втором, если бит не установлен. В более новых версиях используется алгоритм с двумя стрелками для повышения эффективности.

Если подкачка становится слишком частой, свопер выгружает процессы. Сначала выбираются те, что неактивны более 20 секунд, затем — крупнейшие, дольше всех находящиеся в памяти. Для повторной загрузки процессов используется та же логика: приоритет отдается давно выгруженным, но не слишком большим процессам. Загружаются только структура пользователя и таблицы страниц, остальные сегменты подгружаются по мере обращения.

В System V подкачка похожа, но применяется алгоритм часов с одной стрелкой: страница удаляется лишь после нескольких проходов без использования. Вместо одного порога `lotsfree` используются два: `min` и `max`, что уменьшает частую активацию демона и стабилизирует систему.

22. Раскройте сущность межпроцессного обмена и синхронизации в Windows.

Для реализации межпроцессорного обмена с синхронизации в Windows могут использоваться:

1. **Каналы**. Работают в 2 режимах: байтовый режим и режим сообщений (выбирается во время создания). В байтовом режиме каналы работают так же, как и в UNIX. Каналы в режиме сообщений немного похожи на них, но сохраняют границы сообщений — так что четыре записи по 128 байт будут прочитаны как четыре сообщения по 128 байт (а не как одно сообщение размером 512 байт, как это может случиться с каналом в байтовом режиме).
2. **Именованные каналы**. Работают аналогично обычным каналам, но могут также быть использованы в сети.
3. **Почтовые слоты**. Пришли из OS/2, похожи на каналы, но:
   a. Односторонние
   b. Могут быть использованы в сети, без гарантии доставки
   c. Позволяют транслировать сообщение множеству получателей
4. **Сокеты (Sockets)**. Схожи с каналами. Используются для передачи информации между разными процессами как на разных машинах, так и на одной. RPC (Удаленный вызов процедур) – позволяют процессу А «попросить» процесс В вызвать процедуру в его адресном пространстве и вернуть результат. Реализуются поверх транспортного уровня (TCP/IP или ALPC – Advanced Local Procedures Call). ALPC – это средство передачи сообщений в исполняющем уровне режима ядра. Оно оптимизировано для обмена между процессами локального компьютера и не работает по сети. Основная идея — посылать сообщения, которые генерируют ответы.
5. **Объекты**. При помощи объектов сегментов, соответствующим отображаемым на них файлам, можно производить обмен информации между разными процессами.

23. Дайте сравнительную характеристику методов размещения адресов блоков в ufs и ext2.

UFS (Unix File System)

Размещение блоков:

Индексы блоков: UFS использует индексы блоков, хранящиеся в i-узлах (i-nodes). Каждый i-узел содержит прямые и косвенные указатели на блоки данных. 10 прямых и 3 косвенных дисковых адресовю

Прямые указатели: Прямые указатели непосредственно указывают на блоки данных. Длина адресов 3 байта.

Косвенные указатели: Система использует одноуровневые, двухуровневые и трехуровневые косвенные указатели для адресации больших файлов. Это позволяет обращаться к файлам, превышающим размер, который может быть описан только прямыми указателями.

Фрагментация и эффективность:

Группы цилиндров: UFS разделяет диск на группы цилиндров, каждая из которых

имеет свой суперблок, i-узлы и блоки данных. Это позволяет уменьшить фрагментацию и ускорить доступ к данным, так как блоки данных файла располагаются в той же группе цилиндров, что и i-узел.

Два размера блоков: Для увеличения эффективности UFS использует блоки двух размеров, что позволяет оптимально работать как с большими, так и с маленькими файлами.

Ext2 (Second Extended File System)

Размещение блоков:

Индексы блоков: Ext2 также использует i-узлы, которые хранят указатели на блоки данных. В i-узле Ext2 хранится больше прямых указателей (12) по сравнению с UFS, что позволяет более эффективно адресовать небольшие файлы без необходимости использования косвенных указателей. 12 прямых и 3 косвенных дисковых адреса

Косвенные указатели: Как и в UFS, используются одноуровневые, двухуровневые и трехуровневые косвенные указатели для работы с большими файлами. Увеличенная длина адресов до 4 байт позволяет поддерживать более крупные дисковые разделы (до 16 Гбайт).

Фрагментация и эффективность:

Группы блоков: Ext2 делит диск на группы блоков, каждая из которых имеет свой суперблок и описатель группы. Такая структура помогает распределять файлы и каталоги равномерно по всему диску, уменьшая фрагментацию и увеличивая производительность.

Размеры блоков: В отличие от UFS, Ext2 использует блоки фиксированного размера (1 Кбайт), что упрощает управление, но может быть менее эффективно для очень больших файлов.

Сравнение

Прямые указатели: Ext2 имеет больше прямых указателей в i-узле (12 против 10 в UFS), что позволяет быстрее обращаться к небольшим файлам.

Группы данных: UFS использует группы цилиндров для уменьшения фрагментации, тогда как Ext2 использует группы блоков с аналогичной целью, но с различной организацией.

Косвенные указатели: Оба метода используют многоуровневые косвенные указатели, что позволяет эффективно работать с большими файлами.

Размеры блоков: UFS поддерживает блоки двух размеров для оптимизации хранения файлов разных размеров, в то время как Ext2 использует блоки фиксированного размера.

24. Опишите организацию режима ядра в Windows и функции уровня HAL.

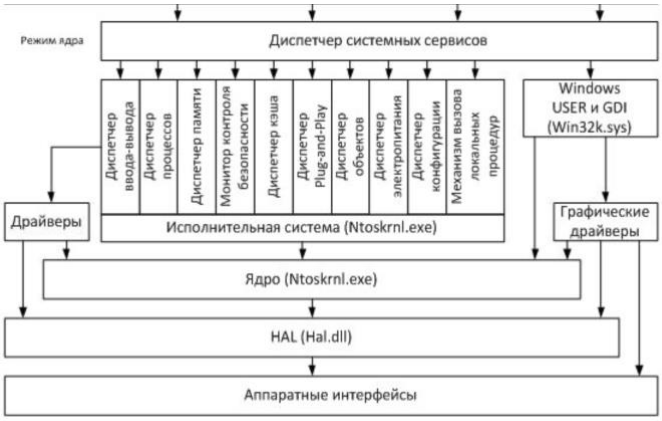

Рис. 11. Организация режима ядра в Windows.

Диспетчер системных сервисов (System Service Dispatcher) работает в режиме ядра, перехватывает вызовы функций от Ntdll.dll, проверяет их параметры и вызывает соответствующие функции из Ntoskrnl.exe.

Исполнительная система и ядро содержатся в Ntoskrnl.exe (NT Operating System Kernel – ядро операционной системы NT) (по поводу использования термина "ядро" в Windows см. лекцию 1 "Введение в операционные системы").

Исполнительная система (Executive) представляет собой совокупность компонентов (называемых диспетчерами – manager), которые реализуют основные задачи операционной системы:

- диспетчер процессов (process manager) – управление процессами и потоками
- диспетчер памяти (memory manager) – управление виртуальной памятью и отображение её на физическую (см. лекцию 8 "Управление памятью");
- монитор контроля безопасности (security reference monitor) – управление безопасностью
- диспетчер ввода вывода (I/O manager), диспетчер кэша (cache Manager), диспетчер Plug and Play (PnP Manager) – управление внешними устройствами и файловыми системами (см. лекцию 10 "Управление устройствами" и лекцию 11 "Файловая система NTFS");
- диспетчер электропитания (power manager) – управление электропитанием и энергопотреблением;
- диспетчер объектов (object manager), диспетчер конфигурации (configuration manager), механизм вызова локальных процедур (local procedure call) – управление служебными процедурами и структурами данных, которые необходимы остальным компонентам.

Ядро (Kernel) содержит функции, обеспечивающие поддержку компонентам исполнительной системы и осуществляющие планирование потоков (см. лекцию 7 "Планирование потоков"), механизмы синхронизации, обработку прерываний.

Компонент Windows USER и GDI отвечает за пользовательский графический интерфейс (окна, элементы управления в окнах – меню, кнопки и т. п., рисование), является частью подсистемы Windows и реализован в драйвере Win32k.sys.

Взаимодействие диспетчера ввода вывода с устройствами обеспечивают драйверы (drivers) – программные модули, работающие в режиме ядра, обладающие максимально полной информацией о конкретном устройстве (драйверы подробнее рассматриваются в лекции 10 "Управление устройствами").

Однако, и драйверы, и ядро не взаимодействуют с физическими устройствами напрямую – посредником между программными компонентами режима ядра и аппаратурой является HAL (Hardware Abstraction Layer) – уровень абстрагирования от оборудования, реализованный в Hal.dll. HAL позволяет скрыть от всех программных компонентов особенности аппаратной платформы (например, различия между материнскими платами), на которой установлена операционная система.

Задача HAL — представить остальной части операционной системы некое абстрактное оборудование, которое скрывает специфичные подробности (версию процессора, чипсет и прочие особенности конфигурации). Эти абстракции уровня HAL представлены в форме независимых от компьютера служб (вызовов процедур и макросов), которые могут использоваться драйверами и NTOS. При использовании служб HAL и отсутствии прямых обращений к оборудованию для драйверов и ядра требуется меньше изменений при переносе на новые процессоры — и почти во всех случаях они могут работать без всякой модификации на всех системах с одинаковой архитектурой процессора (несмотря на разные их версии и разные чипсеты). HAL не обеспечивает абстракций или служб для конкретных устройств ввода-вывода, таких как клавиатура, мышь, диски или устройство управления памятью.

25. Объясните иерархию уровней прикладного программирования в Windows. Приведите примеры собственных вызовов интерфейса прикладного программирования NT.

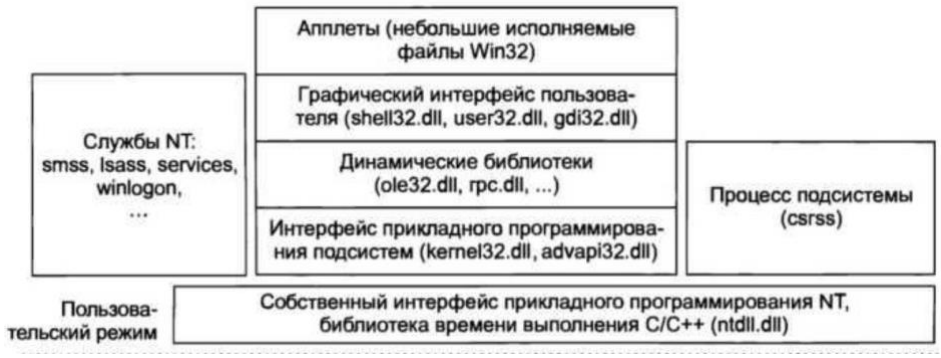

Рис. 12. Иерархия уровней прикладного программирования в Windows.

Под уровнями апплетов и графического интерфейса пользователя находятся интерфейсы программирования, на которых построены приложения. Эти интерфейсы построены на основе более низкоуровневых API, предоставляемых операционной системой. Как и в других операционных системах, они состоят в основном из библиотек кода (DLL), которые программы динамически используют для доступа к функциональным возможностям операционной системы. В Windows имеется также несколько интерфейсов программирования, которые реализованы как работающие в виде отдельных процессов службы. Ядром операционной системы NT является программа режима ядра NTOS (ntoskrnl.exe), которая обеспечивает традиционные интерфейсы системных вызовов (на которых построена остальная часть операционной системы). В Windows только программисты компании Microsoft пишут на уровне системных вызовов. Опубликованные интерфейсы пользовательского режима реализованы при помощи работающих поверх уровней NTOS подсистем «API». Рассмотрим интерфейс прикладного программирования NT для системных вызовов. Первоначально NT поддерживала три «API»: OS/2, POSIX и Win32.

Client/Server Runtime Subsystem (CSRSS), которая обрабатывает консольные окна, создание процессов и потоков, а также взаимодействует с библиотеками Win32 (kernel32.dll, user32.dll, gdi32.dll) для выполнения большинства задач без обращения к ядру. Вызовы к оконному менеджеру и сервисам GDI обрабатываются драйверами режима ядра (win32k.sys) напрямую.

Программы пользовательского режима взаимодействуют с вызовами через опубликованные API, такие как Win32, которые скрывают детали реализации и предоставляют упрощенные интерфейсы для разработчиков приложений.

Собственные вызовы интерфейса прикладного программирования NT (реализованы на уровне исполняющего модуля NTOS):

- NtCreateThread – создать поток в процессе, указанном в *ProcHandle*
- NtAllocateVirtualMemory
- NtMapViewOfSection
- NtReadVirtualMemory
- NtWriteVirtualMemory
- NtCreateFile
- NtDuplicateObject

26. Объясните на примере вызовов Win32 API реализацию программной совместимости Windows-приложений. Объясните иерархию корневых ключей и охарактеризуйте основные подключи реестра в Windows.

Приверженность **обратной совместимости** является одним из ключевых столпов операционной системы Windows, что позволяет старым приложениям работать на новых версиях ОС, даже если они были разработаны десятилетия назад. Эта совместимость во многом достигается за счет тщательно продуманной работы с Win32 API и механизмов реестра Windows.

Программная совместимость Windows-приложений и Win32 API

Win32 API (Application Programming Interface) — это набор функций, структур данных и процедур, которые позволяют приложениям взаимодействовать с операционной системой Windows. Практически каждое действие, которое выполняет приложение в Windows (от создания окна до записи файла или сетевого запроса), использует вызовы Win32 API.

Microsoft прилагает огромные усилия для сохранения поведения функций Win32 API от версии к версии Windows. Например, функция `CreateFile` (используемая для создания или открытия файлов), введенная в ранних версиях Windows NT, до сих пор работает практически так же на Windows 11. Если бы Microsoft изменила ее поведение без предоставления альтернатив, бесчисленные приложения перестали бы работать.

Однако, изменения в операционной системе, такие как новые модели безопасности, изменения в структуре файловой системы или появление новых технологий, могут нарушить работу старых приложений, которые были написаны без учета этих изменений. Для решения таких проблем Windows использует различные механизмы совместимости, наиболее значимыми из которых являются **прослойки совместимости (Compatibility Shims)**.

Пример реализации совместимости с помощью Compatibility Shims:

Представьте старое приложение, разработанное для Windows XP, которое запускается только с правами администратора и пытается записывать данные напрямую в папку "Program Files", что в современных версиях Windows (начиная с Vista) запрещено из-за ужесточения безопасности (контроль учетных записей пользователей, UAC). Без механизма совместимости это приложение просто выдаст ошибку и завершится.

Здесь в дело вступают **shims**. Shim — это небольшая библиотека, которая перехватывает вызовы Win32 API приложения и изменяет их поведение до или после фактического вызова системной функции. Это происходит прозрачно для приложения.

- **Пример shim: `ForceAdminAccess` (Принудительный доступ администратора):**
  - **Проблема:** Старое приложение проверяет, запущен ли оно от имени администратора, используя устаревшие или специфичные для Windows XP методы. На современных ОС без административных прав оно отказывается работать.
  - **Решение `ForceAdminAccess` shim:** Когда приложение вызывает определенную функцию Win32 API для проверки своих прав (например, `IsUserAnAdmin` или более низкоуровневые API безопасности), shim перехватывает этот вызов. Вместо того чтобы позволить ОС ответить "нет" (если пользователь не администратор), shim **фальсифицирует ответ** и возвращает "да", убеждая приложение, что оно запущено с административными правами. Таким образом, приложение запускается, даже если пользователь на самом деле не обладает полными правами администратора.
- **Пример shim: `CorrectFilePaths` (Коррекция путей к файлам):**
  - **Проблема:** Приложение пытается записывать данные в папку "Program Files" или "Windows" (например, ini-файлы или логи), что запрещено UAC.
  - **Решение `CorrectFilePaths` shim (виртуализация UAC):** Когда приложение

вызывает Win32 API для записи файла (например, `WriteFile` или `CreateFile`) в защищенную системную папку, shim перехватывает этот вызов. Вместо того чтобы разрешить запись в оригинальную папку (что приведет к отказу в доступе), shim **перенаправляет операцию записи** в специальную, доступную для записи папку в профиле пользователя (например, `C:\Users\<username>\AppData\Local\VirtualStore\Program Files\<appname>`). Приложение "думает", что успешно записало файл в "Program Files", но на самом деле файл находится в виртуальном хранилище. При чтении того же файла shim также перехватывает вызов и сначала проверяет виртуальное хранилище, а затем, если файл там не найден, оригинальное местоположение.

Эти shims "подставляют" себя между приложением и операционной системой, перехватывая вызовы Win32 API и модифицируя их в реальном времени. Это позволяет старым приложениям корректно работать в новых средах без изменения их кода. Механизмы совместимости также включают **блокировки (Blocks)**, которые предотвращают запуск приложений, если они заведомо вызывают серьезные проблемы безопасности или стабильности.

Иерархия корневых ключей и основные подключи реестра Windows

Реестр Windows (Windows Registry) — это централизованная, иерархическая база данных, используемая операционной системой для хранения низкоуровневых настроек системного оборудования, установленных приложений, профилей пользователей и настроек операционной системы. Он заменил устаревшие INI-файлы, предлагая более надёжное и структурированное хранение данных, поддержку сетевых сред, более эффективный доступ и централизованное резервное копирование.

Реестр организован в виде древовидной структуры, состоящей из **ключей** (аналогичных папкам), которые могут содержать **подключи** и **значения** (аналогичные файлам). Значения хранят фактические данные и имеют различные типы (строки, числа, бинарные данные и т.д.).

Корневые ключи (Root Keys) реестра:

В реестре Windows существует пять предопределенных корневых ключей (также известных как кусты — Hives), каждый из которых начинается с `HKEY_`. Эти ключи являются логическими точками входа и не существуют как файлы на диске напрямую, а скорее представляют собой динамические представления данных из различных файлов реестра.

1. **`HKEY_CLASSES_ROOT (HKCR)`**
   - **Назначение:** Содержит информацию о зарегистрированных приложениях, типы файлов (расширения и связанные с ними программы), объекты COM (Component Object Model) и OLE (Object Linking and Embedding).
   - **Характеристика:** Является логическим объединением двух подключей: `HKEY_LOCAL_MACHINE\SOFTWARE\Classes` (системные настройки для всех пользователей) и `HKEY_CURRENT_USER\SOFTWARE\Classes` (пользовательские настройки, которые могут переопределять системные).
   - **Основные подключи:**
     - `.txt`, `.docx`, `.exe` и т.д.: Определяют, какая программа должна открывать файлы с соответствующим расширением.
     - `CLSID`: Содержит идентификаторы COM-классов (Class IDs) и пути к их библиотекам (DLL/EXE).
     - `Interface`: Содержит идентификаторы COM-интерфейсов (Interface IDs).
     - `ProgID`: Содержит "человекочитаемые" идентификаторы для COM-объектов.
2. **`HKEY_CURRENT_USER (HKCU)`**
   - **Назначение:** Содержит настройки, специфичные для текущего вошедшего в систему пользователя. Эти настройки загружаются при входе пользователя в

систему.
   - **Характеристика:** Является динамическим подключом, который указывает на ветку профиля текущего пользователя в `HKEY_USERS` (например, `HKEY_USERS\<SID_пользователя>`).
   - **Основные подключи:**
     - `AppEvents`: Настройки звуковых схем.
     - `Control Panel`: Персонализированные настройки панели управления (например, параметры рабочего стола, мыши, клавиатуры).
     - `Environment`: Переменные среды для текущего пользователя.
     - `Keyboard Layout`: Настройки раскладки клавиатуры.
     - `Software`: Настройки программного обеспечения, установленного для текущего пользователя.
     - `Printers`: Настройки принтеров текущего пользователя.
3. **`HKEY_LOCAL_MACHINE (HKLM)`**
   - **Назначение:** Содержит настройки, специфичные для компьютера в целом и применимые ко всем пользователям.
   - **Характеристика:** Включает в себя системные настройки, информацию об аппаратном обеспечении, драйверах, службах и общесистемные настройки программного обеспечения.
   - **Основные подключи:**
     - `HARDWARE`: Информация об обнаруженном оборудовании компьютера. Генерируется каждый раз при загрузке системы.
     - `SAM (Security Accounts Manager)`: База данных локальных учетных записей пользователей и групп. Защищен от прямого доступа.
     - `SECURITY`: Локальная политика безопасности, привилегии пользователей. Также защищен.
     - `SOFTWARE`: Настройки программного обеспечения, установленного для всех пользователей системы.
       - Например, `Microsoft`: содержит настройки для продуктов Microsoft.
       - `WOW6432Node`: Для 32-битных приложений на 64-битных системах.
     - `SYSTEM`: Данные о конфигурации системы, загрузочные данные, список драйверов устройств и служб.
       - `CurrentControlSet`: Содержит текущую конфигурацию системы.
4. **`HKEY_USERS (HKU)`**
   - **Назначение:** Содержит все активно загруженные профили пользователей на компьютере.
   - **Характеристика:** Каждый под ключ в `HKEY_USERS` назван по **SID (Security Identifier)** пользователя, который представляет собой уникальный идентификатор для каждой учетной записи пользователя или группы. `HKEY_CURRENT_USER` является ссылкой на один из этих SID-ключей.
   - **Основные подключи:**
     - `.DEFAULT`: Настройки по умолчанию для новых пользователей.
     - `<SID_пользователя>`: Содержит полный набор настроек для конкретного пользователя, аналогично структуре `HKEY_CURRENT_USER` для этого пользователя.
5. **`HKEY_CURRENT_CONFIG (HKCC)`**
   - **Назначение:** Содержит текущую аппаратную конфигурацию профиля, которая используется системой при запуске.
   - **Характеристика:** Является динамической ссылкой на ветвь `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Hardware Profiles\Current`.
   - **Основные подключи:**
     - `System`: Настройки, специфичные для текущей аппаратной конфигурации, такие как параметры дисплея, принтеров и других устройств, которые могут

изменяться в зависимости от профиля оборудования.

27. Объясните принцип работы в Windows отложенных вызовов процедур. Охарактеризуйте структуру заголовка диспетчеризации.

Отложенные вызовы процедур (Deferred Procedure Calls, **DPC**) являются фундаментальным механизмом в ядре Windows, позволяющим операционной системе и драйверам устройств выполнять несрочную работу прерываний на более низком уровне приоритета (IRQL), тем самым повышая общую отзывчивость системы.

Принцип работы отложенных вызовов процедур (DPC)

Работа DPC основана на разделении обработки аппаратных прерываний на две части:

1. **Высокоприоритетная часть (первичная обработка прерывания):** Когда происходит аппаратное прерывание (например, от сетевой карты или дискового контроллера), процессор немедленно переключается на выполнение процедуры обслуживания прерывания (Interrupt Service Routine, **ISR**). ISR работает на высоком уровне IRQL (уровне прерывания устройства) и должна быть максимально короткой и быстрой. Её основная задача — подтвердить прерывание оборудованию, собрать минимально необходимую информацию и, если есть несрочная работа, поставить её в очередь на выполнение с помощью DPC.
2. **Низкоприоритетная часть (вторичная обработка прерывания):** После того как ISR завершает свою работу и IRQL процессора опускается до уровня **DISPATCH_LEVEL**, Windows проверяет наличие DPC в очереди. Если DPC присутствуют, система начинает их выполнение. DPC-подпрограммы выполняются на DISPATCH_LEVEL, что означает, что они имеют более низкий приоритет, чем аппаратные прерывания, но более высокий, чем обычные потоки пользовательского режима и большинство потоков ядра.

Механизм очереди и выполнения:

- **Очередь DPC:** Каждому процессору в системе соответствует своя очередь DPC. Когда ISR или другая подпрограмма ядра вызывает функцию `KeInsertQueueDpc`, соответствующий объект DPC помещается в очередь DPC текущего процессора.
- **Диспетчеризация DPC:** Специальная часть ядра Windows, называемая **диспетчером DPC**, отвечает за выполнение DPC. Когда IRQL процессора падает до DISPATCH_LEVEL, диспетчер DPC просматривает очередь DPC и выполняет зарегистрированные DPC-подпрограммы одну за другой.
- **Контекст выполнения:** DPC-подпрограммы выполняются в произвольном потоковом контексте, то есть они не привязаны к конкретному пользовательскому или системному потоку. Это означает, что DPC не могут выполнять операции, которые требуют контекста потока (например, доступ к выгружаемой памяти, ожидание объектов диспетчера).

Назначение и сценарии использования:

- **Уменьшение времени работы ISR:** DPC позволяют ISRам быстро завершать работу, освобождая процессор для обработки других, более критичных прерываний.
- **Отложенная обработка данных:** Используются для обработки больших объемов данных, полученных от устройств, или для выполнения операций, которые не критичны по времени и не требуют высокого IRQL (например, завершение I/O запросов).
- **Асинхронные события:** Подходят для обработки асинхронных событий, будь то от аппаратного обеспечения или от других частей ядра.
- **Таймеры:** Таймеры ядра часто используют DPC для выполнения своих колбэк-функций по истечении заданного времени.

Структура заголовка диспетчеризации DPC (KDPC)

Каждый DPC представлен структурой **_KDPC** (или просто KDPC) в ядре Windows. Эта структура содержит всю необходимую информацию для диспетчеризации и выполнения DPC-подпрограммы. Важно отметить, что KDPC является внутренней, непрозрачной структурой ядра, и Microsoft не рекомендует напрямую манипулировать её полями, вместо этого предоставляя специальные функции (например, KeInitializeDpc, KeInsertQueueDpc).

Типичная (упрощенная) структура _KDPC может выглядеть следующим образом (поля могут немного отличаться в разных версиях Windows):

```
typedef struct _KDPC {
    // В старых версиях Windows: LIST_ENTRY DpcListEntry;
    // В Windows 8.1 и новее:
    SINGLE_LIST_ENTRY DpcListEntry; // Запись для связного списка очереди DPC

    KDEFERRED_ROUTINE *DeferredRoutine; // Указатель на DPC-подпрограмму
    PVOID DeferredContext;              // Контекст, передаваемый DPC-подпрограмме
    PVOID SystemArgument1;              // Дополнительный аргумент 1
    PVOID SystemArgument2;              // Дополнительный аргумент 2

    // Ниже приведены дополнительные поля, которые могут присутствовать
    // и используются для управления состоянием и свойствами DPC:
    UCHAR Type;                         // Тип DPC (обычный, таймерный и т.д.)
    UCHAR Importance;                   // Важность DPC (обычный, высокий, низкий)
    USHORT Number;                      // Номер процессора, к которому привязан DPC (для целевого
DPC)
    BOOLEAN Executing;                  // Флаг, указывающий, выполняется ли DPC
    // ... и другие поля, связанные с состоянием и флагами.
} KDPC, *PKDPC;

```

Характеристика основных полей заголовка диспетчеризации:

- **DpcListEntry**: Это поле используется для связывания объектов KDPC в очередь DPC. В более старых версиях Windows это была структура LIST_ENTRY для двусвязного списка, но начиная с Windows 8.1, для оптимизации производительности DPC-очередей, она была изменена на SINGLE_LIST_ENTRY для односвязного списка.
- **DeferredRoutine**: Это указатель на саму DPC-подпрограмму (функцию), которую должно вызвать ядро, когда настанет очередь DPC. Эта функция имеет следующую сигнатуру:

```
VOID
DpcForIsr(
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
);

```

- **DeferredContext**: Указатель на контекстные данные, которые будут переданы в DeferredRoutine в качестве первого аргумента. Это позволяет DPC-подпрограмме получать доступ к информации, необходимой для выполнения своей работы (например, указатель на расширение устройства или I/O запрос).
- **SystemArgument1** и **SystemArgument2**: Дополнительные аргументы, которые также передаются в DeferredRoutine. Они часто используются для передачи специфических данных, собранных ISR, или для флагов, управляющих поведением DPC.
- **Type**: Определяет тип DPC. Например, это может быть стандартный DPC, DPC, связанный с таймером, или DPC для целевого процессора.
- **Importance**: Указывает на важность DPC для диспетчера. DPC с высокой важностью могут быть обработаны быстрее.
- **Number**: В случае целевых DPC, это поле может указывать номер процессора, на

котором должен быть выполнен DPC.

Преимущества и ограничения DPC

Преимущества:

- **Быстрое завершение ISR:** Позволяют ISRам завершать работу максимально быстро, минимизируя время блокировки аппаратных прерываний.
- **Повышенная отзывчивость системы:** Переносят несрочную работу на более низкий IRQL, что позволяет ядру обрабатывать другие критические прерывания и планировать потоки.
- **Унифицированный механизм:** Предоставляют стандартизированный способ отложенной обработки событий в ядре.

Ограничения:

- **Выполнение на DISPATCH_LEVEL:** DPC-подпрограммы выполняются на DISPATCH_LEVEL, что накладывает серьезные ограничения:
  - **Запрет на доступ к выгружаемой памяти:** DPC не могут обращаться к памяти, которая может быть выгружена на диск (pageable memory), так как это может привести к ошибкам страницы на DISPATCH_LEVEL. Вся используемая память должна быть заблокирована (nonpaged pool).
  - **Запрет на ожидание объектов диспетчера:** DPC не могут вызывать функции, которые могут перевести их в состояние ожидания (например, `KeWaitForSingleObject`), так как это может привести к взаимоблокировке (deadlock) системы.
  - **Короткое время выполнения:** DPC должны выполняться очень быстро (желательно менее 100 микросекунд). Длительное выполнение DPC может привести к задержкам в обработке других DPC, аппаратных прерываний, а также к "фризам" системы, задержкам аудио/видео и падениям (BSOD) из-за срабатывания DPC Watchdog Timer.
- **Синхронизация:** При доступе к общим данным DPC должны использовать **спин-блокировки (spin locks)**, так как они не могут использовать мьютексы или другие объекты диспетчера.
- **Произвольный контекст потока:** Поскольку DPC выполняются в произвольном контексте, они не могут полагаться на свойства текущего потока (например, на его приоритет или права доступа).

28. Раскройте сущность процесса управления диспетчером объектов Windows объекта исполняющей системы. Охарактеризуйте структуры данных таблицы описателей.

Диспетчер Объектов (Object Manager) в Windows является одним из ключевых компонентов исполнительной системы (Executive), отвечающим за создание, управление, именование и защиту всех системных ресурсов, представленных в виде объектов. Он предоставляет единый, консистентный интерфейс для взаимодействия с этими ресурсами, абстрагируя детали их реализации и обеспечивая безопасность доступа. По сути, Диспетчер Объектов — это центральный узел, через который все подсистемы и приложения запрашивают доступ к системным ресурсам.

Основные функции диспетчера объектов:

1. **Создание и Удаление Объектов (Object Creation & Deletion):**
   - Диспетчер Объектов отвечает за выделение памяти для нового объекта, инициализацию его заголовка и тела, а также регистрацию объекта в системе.

- При удалении объекта он управляет освобождением выделенных ресурсов.

2. **Управление Идентификаторами (Handles):**
   - Вместо прямого доступа к объектам, процессы получают числовые идентификаторы, называемые **описателями (handles)**. Диспетчер Объектов управляет созданием, дублированием и закрытием этих описателей.
   - Каждый описатель действителен только в контексте того процесса, который его получил.
3. **Именование Объектов (Object Naming):**
   - Диспетчер Объектов поддерживает иерархическое пространство имен (namespace), подобное файловой системе, где объекты могут быть доступны по имени (например, `\Device\HarddiskVolume1\Windows`). Это позволяет процессам находить и открывать существующие объекты.
   - Объекты могут быть как именованными (доступными в пространстве имен), так и неименованными (доступными только через передачу описателя).
4. **Безопасность Объектов (Object Security):**
   - Диспетчер Объектов является точкой принудительного применения безопасности для всех объектов. Он использует **дескрипторы безопасности (security descriptors)**, прикрепленные к каждому объекту, для определения того, каким пользователям или группам разрешен или запрещен доступ к объекту и какие операции они могут выполнять.
   - При каждой попытке процесса получить доступ к объекту (например, открыть его или выполнить операцию), Диспетчер Объектов взаимодействует с **Монитором Ссылок Безопасности (Security Reference Monitor - SRM)**. SRM принимает решение о доступе на основе дескриптора безопасности объекта и токена доступа запрашивающего процесса. Если доступ разрешен, SRM возвращает решение Диспетчеру Объектов, который затем предоставляет доступ. Если доступ запрещен, запрос блокируется, и возвращается ошибка.
5. **Управление Ссылками (Reference Counting):**
   - Каждый объект имеет счетчик ссылок (reference count). Когда процесс открывает объект или дублирует описатель, счетчик увеличивается. Когда процесс закрывает описатель, счетчик уменьшается.
   - Объект удаляется из памяти только тогда, когда его счетчик ссылок достигает нуля, что гарантирует, что объект не будет удален, пока он используется каким-либо процессом.
6. **Расширяемость (Extensibility):**
   - Диспетчер Объектов является расширяемым, позволяя различным исполнительным подсистемам (например, Диспетчеру Памяти, Диспетчеру Процессов) определять свои собственные типы объектов и регистрировать их у Диспетчера Объектов. Это позволяет единообразно управлять широким спектром системных ресурсов.

Взаимосвязь описателей, объектов и пространства имен:

- **Объект:** Это экземпляр системного ресурса (например, процесса, потока, файла, события, мьютекса). Объекты существуют в ядре операционной системы.
- **Описатель (Handle):** Это непрозрачное целое число, которое процесс использует для ссылки на объект. Описатель является локальным для каждого процесса. Процесс не работает напрямую с указателем на объект в памяти ядра; вместо этого он использует описатель, который служит индексом в его таблице описателей.
- **Таблица Описателей (Handle Table):** Каждому процессу назначается своя таблица описателей, которая является массивом или деревом записей. Каждая запись в этой таблице сопоставляет числовой описатель (который процесс использует) с фактическим указателем на объект ядра и информацией о правах доступа.
- **Пространство Имен (Namespace):** Диспетчер Объектов поддерживает иерархическую

структуру каталогов и объектов. Процессы могут открыть объект по его имени в этом пространстве имен (например, `\BaseNamedObjects\MyMutex`), что приводит к получению нового описателя к существующему ядру объекта. Если объект не имеет имени, он может быть доступен только путем передачи описателя между процессами (например, через наследование при создании дочернего процесса или с помощью функции `DuplicateHandle`).

Таким образом, Диспетчер Объектов обеспечивает упорядоченный, безопасный и масштабируемый способ управления доступом к системным ресурсам, используя описатели как механизм абстракции и пространство имен для организации объектов.

Характеристика структур данных таблицы описателей

Каждый процесс в Windows имеет свою собственную **Таблицу Описателей (Handle Table)**, которая является центральной структурой данных, управляемой Диспетчером Объектов для отслеживания объектов, открытых этим процессом. Эта таблица является мостом между логическим представлением ресурсов для пользовательского процесса (описатель как целое число) и их физическим представлением в ядре (объект ядра).

Таблица описателей представляет собой структуру, которая эффективно сопоставляет числовые описатели, используемые процессом, с соответствующими объектами ядра и связанными с ними правами доступа. Внутренне она реализована как дерево или массив записей для обеспечения производительности.

Каждая активная запись в таблице описателей обычно представляет собой пару, содержащую информацию, необходимую для работы с объектом:

1. **Указатель на Объект Ядра (Object Pointer):**
   - Это фактический адрес в памяти ядра, указывающий на структуру **Заголовка Объекта (Object Header)** соответствующего объекта. Заголовок объекта содержит общую информацию об объекте, такую как его тип, счетчик ссылок и дескриптор безопасности.
   - Через этот указатель Диспетчер Объектов получает доступ к телу объекта (которое специфично для его типа — например, структура PEB для объекта процесса, EPROCESS), счетчику ссылок и другим метаданным.
2. **Маска Предоставленных Прав Доступа (Granted Access Mask):**
   - Это битовая маска, указывающая конкретные права доступа, которые были предоставлены процессу для этого конкретного объекта при его открытии.
   - Например, для файлового объекта это может быть `FILE_READ_DATA`, `FILE_WRITE_DATA`, `FILE_EXECUTE` и т.д. Для процесса это может быть `PROCESS_TERMINATE`, `PROCESS_VM_READ` и т.д.
   - Эта маска является результатом проверки безопасности, выполненной SRM в момент открытия объекта. Диспетчер Объектов использует эту маску для быстрой проверки прав доступа при последующих операциях с объектом через этот описатель, избегая повторной полной проверки безопасности с SRM.
3. **Атрибуты Описателя (Handle Attributes):**
   - Каждая запись также может содержать флаги или атрибуты, которые определяют поведение описателя. Наиболее распространенными из них являются:
     - **`OBJ_INHERIT` (наследуемый):** Если этот флаг установлен, дочерние процессы, созданные этим процессом, автоматически наследуют этот описатель.
     - **`OBJ_PROTECT_FROM_CLOSE` (защита от закрытия):** Этот флаг, если установлен, предотвращает закрытие описателя с помощью `CloseHandle`. Это используется для критически важных описателей, которые не должны быть случайно закрыты.
     - *Возможны и другие атрибуты в зависимости от версии Windows и конкретного типа объекта.*

29. Объясните принцип выполнения процедур объекта Windows, предоставляемые при определении нового типа объектов. Перечислите основные типы объектов исполняющей системы и объясните их назначение.

Windows, как объектно-ориентированная операционная система, использует концепцию объектов ядра для управления системными ресурсами. Когда речь идет о **процедурах объекта, предоставляемых при определении нового типа объектов**, имеется в виду набор функций обратного вызова (callback functions), которые система вызывает в ответ на определенные операции с объектами этого типа.

Принцип выполнения процедур объекта при определении нового типа объектов:

При создании нового типа объектов в ядре Windows (это обычно делается разработчиками драйверов или системных компонентов, а не обычными приложениями), необходимо зарегистрировать у Менеджера Объектов (Object Manager) набор процедур, которые будут выполняться при различных операциях над экземплярами этого типа объекта. Эти процедуры обеспечивают "поведение" нового типа объекта.

Основные процедуры (методы), которые могут быть зарегистрированы для нового типа объектов:

- **Open (Открыть):** Вызывается каждый раз, когда создается новый дескриптор для объекта (например, при вызове функции CreateFile для файлового объекта). Эта процедура используется редко для большинства типов объектов, но может быть полезна для инициализации специфических состояний, связанных с открытием объекта.
- **Parse (Разобрать/Распарсить):** Используется для типов объектов, которые расширяют пространство имен, таких как файлы или ключи реестра. Эта процедура отвечает за "разбор" имени объекта, чтобы найти его в иерархической структуре пространства имен.
- **Close (Закрыть):** Вызывается, когда закрывается последний дескриптор объекта. Это может потребоваться для выполнения действий по очистке состояния, видимых побочных эффектов или освобождения временных ресурсов, связанных с открытыми дескрипторами.
- **Delete (Удалить):** Вызывается, когда из объекта удаляется последняя ссылка (указатель). Это означает, что объект больше не используется и готов к удалению из памяти. Эта процедура отвечает за окончательную деактивацию объекта и освобождение выделенной для него памяти. Close и Delete часто работают в паре: Close очищает состояние, связанное с дескриптором, а Delete освобождает сам объект.
- **Security (Безопасность):** Эта процедура вызывается для получения или установки дескриптора безопасности объекта. Она отвечает за реализацию контроля доступа к объекту.
- **QueryName (Запросить имя):** Используется для получения имени объекта.
- И другие, менее распространенные процедуры, в зависимости от специфики типа объекта.

Таким образом, принцип заключается в том, что Менеджер Объектов, когда приложение выполняет операцию над объектом (например, открывает его, закрывает или пытается получить к нему доступ), автоматически вызывает соответствующую зарегистрированную процедуру, которая была предоставлена при определении типа этого объекта. Это обеспечивает единый и расширяемый механизм управления ресурсами в Windows.

Основные типы объектов исполняющей системы Windows и их назначение:

Исполняющая система Windows (Executive) использует множество типов объектов для управления системными ресурсами. Ниже приведены некоторые из наиболее важных:

1. **Процесс (Process):**

- **Назначение:** Представляет собой экземпляр запущенной программы. Процесс включает в себя адресное пространство, набор ресурсов (дескрипторы открытых файлов, портов и т.д.) и один или несколько потоков.
- **Пример:** Каждый раз, когда вы запускаете приложение (например, Word, браузер), создается новый процесс.

2. **Поток (Thread):**
   - **Назначение:** Является основной единицей выполнения в Windows. Поток — это последовательность инструкций, которые могут выполняться процессором. Каждый процесс содержит хотя бы один поток (основной поток). Потоки внутри одного процесса разделяют его адресное пространство и ресурсы.
   - **Пример:** Внутри одного процесса (например, браузера) могут быть разные потоки, отвечающие за отрисовку веб-страницы, загрузку данных, обработку пользовательского ввода и т.д.
3. **Событие (Event):**
   - **Назначение:** Используется для синхронизации потоков. Событие может находиться в двух состояниях: сигнальное (просигнализированное) или несигнальное (непросигнализированное). Потоки могут ожидать сигнала события, чтобы продолжить выполнение.
   - **Пример:** Один поток завершил определенную задачу и "сигнализирует" событие, чтобы другой поток, ожидающий этого события, мог начать свою работу.
4. **Мьютекс (Mutex):**
   - **Назначение:** Также используется для синхронизации, но предназначен для обеспечения взаимного исключения. Мьютекс позволяет только одному потоку одновременно получить доступ к разделяемому ресурсу. Если один поток владеет мьютексом, другие потоки, пытающиеся получить его, будут заблокированы до тех пор, пока мьютекс не будет освобожден.
   - **Пример:** Защита доступа к глобальной переменной или файлу, чтобы избежать конфликтов при одновременном изменении разными потоками.
5. **Семафор (Semaphore):**
   - **Назначение:** Используется для контроля доступа к ресурсу, который может быть использован несколькими потоками одновременно, но не бесконечным их числом. Семафор имеет счетчик, который уменьшается при захвате ресурса и увеличивается при его освобождении. Поток блокируется, если счетчик равен нулю.
   - **Пример:** Ограничение количества одновременных подключений к базе данных или числа потоков, обрабатывающих задачи в пуле.
6. **Файл (File):**
   - **Назначение:** Представляет собой именованную область хранения данных на диске или другом устройстве ввода-вывода. Объекты файлов позволяют приложениям читать и записывать данные.
   - **Пример:** Документы, исполняемые файлы, изображения.
7. **Сопоставление файлов (File Mapping):**
   - **Назначение:** Позволяет процессу получить прямой доступ к содержимому файла как к области памяти. Это эффективный способ работы с большими файлами и обмена данными между процессами.
   - **Пример:** Использование файла в качестве общей памяти для двух разных приложений.
8. **Труба (Pipe):**
   - **Назначение:** Обеспечивает механизм межпроцессного взаимодействия (IPC), позволяя процессам обмениваться данными. Бывают именованные и анонимные трубы.
   - **Пример:** Передача вывода одной программы в качестве ввода для другой (как в командной строке dir | more).

9. **Устройство (Device):**
   - **Назначение:** Представляет собой физическое или логическое устройство, такое как принтер, сетевая карта, клавиатура или диск. Объекты устройств позволяют системе и драйверам взаимодействовать с аппаратным обеспечением.
   - **Пример:** Объект для принтера, через который приложения отправляют данные на печать.
10. **Маркер доступа (Access Token):**
   - **Назначение:** Содержит информацию о безопасности пользователя или процесса, такую как идентификаторы безопасности (SID) пользователя и групп, а также привилегии. Используется для контроля доступа к объектам и ресурсам.
   - **Пример:** Когда пользователь входит в систему, ему выдается маркер доступа, который определяет его права.
11. **Раздел (Section):**
   - **Назначение:** Представляет собой область памяти, которая может быть использована для обмена данными между процессами. Разделы могут быть связаны с файлами или быть анонимными (только в памяти).
   - **Пример:** Использование общей памяти для высокоскоростного обмена данными между приложениями.

30. Раскройте особенности использования синхронной стратегии при работе с файлами подкачки в Windows. Опишите преимущества и недостатки адресации физической памяти большого объема на примере элементов таблицы страниц различных архитектур.

Файл подкачки (pagefile.sys) в Windows служит для расширения оперативной памяти (RAM), когда физическая память заполнена. Он позволяет операционной системе перемещать данные между RAM и жестким диском, обеспечивая возможность работы с большим количеством приложений одновременно. Синхронная стратегия при работе с файлом подкачки означает, что операции чтения или записи в этот файл выполняются синхронно, т.е. процесс, инициирующий операцию, ожидает ее завершения перед продолжением работы. Это характерно для многих операций ввода-вывода в операционных системах, если не настроено иначе для асинхронного выполнения.

Особенности синхронной стратегии:

1. **Синхронные операции**:
   - При возникновении страничного прерывания (когда требуемая страница отсутствует в оперативной памяти), система инициирует чтение данных из файла подкачки. Процесс приостанавливается до завершения этой операции, что обеспечивает последовательность выполнения.
   - Аналогично, при записи измененных страниц в файл подкачки, операция может быть синхронной, особенно если требуется немедленная фиксация данных, например, для критически важных приложений, таких как базы данных.
2. **Роль системного кэша**:
   - Windows использует системный кэш для буферизации операций с файлом подкачки, что минимизирует прямые обращения к диску и улучшает производительность. Однако при синхронной стратегии кэш может быть обойден, если требуется гарантированное выполнение операции без задержек, связанных с кэшированием.
3. **Приоритет операций**:
   - Синхронные операции с файлом подкачки имеют высокий приоритет, поскольку они связаны с обработкой страничных прерываний, которые критически важны для работы приложений. Это обеспечивает стабильность системы, но может приводить

к задержкам других процессов.

4. **Контроль со стороны приложений**:
   - Приложения могут влиять на поведение синхронных операций через API, такие как FILE_FLAG_WRITE_THROUGH для записи без кэширования или FILE_FLAG_NO_BUFFERING для прямого доступа к диску. Это позволяет разработчикам настраивать поведение в зависимости от требований приложения.

Преимущества синхронной стратегии

- **Предсказуемость**: Синхронные операции гарантируют, что данные будут прочитаны или записаны до продолжения выполнения программы, что важно для приложений, требующих строгой целостности данных, например, в финансовых системах или базах данных.
- **Простота управления**: Отсутствие необходимости в сложной обработке асинхронных обратных вызовов или событий упрощает разработку и отладку программ, особенно для небольших приложений.
- **Надежность**: Для критически важных данных синхронная запись минимизирует риск потери данных при сбоях, что особенно важно в серверных средах.

Недостатки синхронной стратегии

- **Задержки**: Поток блокируется до завершения операции ввода-вывода, что может значительно снижать производительность, особенно при использовании медленных накопителей, таких как HDD. Это особенно заметно в системах с высокой нагрузкой.
- **Ограниченная масштабируемость**: В многопоточных приложениях синхронные операции могут привести к узким местам, так как потоки будут ожидать завершения операций, что ограничивает параллелизм.
- **Высокая нагрузка на диск**: Частые синхронные обращения к файлу подкачки увеличивают износ накопителя, особенно на HDD, и могут сократить срок службы SSD из-за ограниченного числа циклов записи.

Адресация физической памяти большого объема и элементы таблиц страниц

Адресация физической памяти большого объема — это способность системы обращаться к большим объемам физической памяти, например, более 4 ГБ в 32-разрядных системах или терабайты в 64-разрядных. Это достигается с помощью таблиц страниц, которые являются структурами данных, используемыми операционной системой для отображения виртуальных адресов в физические. Таблицы страниц играют ключевую роль в виртуальной памяти, обеспечивая защиту памяти, разделение и возможность использования памяти, превышающей физическую.

Различные архитектуры процессоров, такие как x86 и x86-64, имеют свои подходы к реализации таблиц страниц для поддержки больших объемов памяти. Рассмотрим примеры на архитектурах x86 и x86-64, а также общие преимущества и недостатки.

Архитектура x86 (32-разрядная)

- **Структура таблиц страниц**:
  - Используется двухуровневая структура: каталог страниц (Page Directory) и таблицы страниц (Page Tables).
  - Каждая таблица содержит 1024 записи, что позволяет адресовать до 4 ГБ памяти (2^32).
  - Размер записи в таблице страниц — 4 байта, содержащих физический адрес страницы и атрибуты, такие как доступ на чтение/запись и присутствие в памяти.
- **Расширение PAE (Physical Address Extension)**:
  - Для адресации более 4 ГБ в 32-разрядных системах используется PAE, расширяющий адресное пространство до 36 бит (64 ГБ).
  - Добавляется третий уровень — PDPT (Page Directory Pointer Table), содержащий 4 записи.

- Размер записи увеличивается до 8 байт для поддержки 36-битных физических адресов.
- **Преимущества**:
  - Позволяет 32-разрядным системам использовать больше памяти, чем стандартные 4 ГБ, что было важно для серверов в эпоху 32-битных систем.
  - Обеспечивает совместимость с устаревшими приложениями, не требующими переписывания под 64-битные системы.
- **Недостатки**:
  - Увеличивает накладные расходы на управление таблицами страниц из-за дополнительного уровня PDPT, что может замедлять работу.
  - Ограничивает виртуальное адресное пространство до 4 ГБ для каждого процесса, даже при использовании PAE, что увеличивает зависимость от файла подкачки.
  - Более сложная реализация, что может приводить к дополнительным задержкам при обработке страничных прерываний.

Архитектура x86-64 (64-разрядная)
- **Структура таблиц страниц**:
  - Используется четырехуровневая структура: PML4 (Page Map Level 4), PDPT, Page Directory, Page Table.
  - Каждая таблица содержит 512 записей (по 8 байт), что позволяет адресовать до 2^48 байт (256 ТБ) виртуальной памяти.
  - 64-разрядные адреса обеспечивают поддержку больших объемов физической памяти (до 52 бит в современных процессорах).
- **Преимущества**:
  - Поддержка огромных объемов физической памяти (терабайты) без необходимости специальных расширений, как PAE, что идеально для современных серверов и рабочих станций.
  - Упрощенное управление большими адресными пространствами благодаря 64-разрядным указателям, что снижает зависимость от файла подкачки.
  - Высокая производительность за счет аппаратной поддержки, например, буфера трансляции (TLB), который ускоряет доступ к памяти.
- **Недостатки**:
  - Увеличенные накладные расходы на хранение таблиц страниц (четыре уровня вместо двух в x86), что увеличивает потребление памяти.
  - Более высокое потребление памяти для таблиц страниц, особенно при фрагментации адресного пространства, что может быть проблемой на системах с ограниченными ресурсами.
  - Требует дополнительных механизмов для совместимости с 32-разрядными приложениями, таких как WOW64, что добавляет сложность.

Общие преимущества и недостатки адресации большого объема памяти
- **Преимущества**:
  - Возможность использования больших объемов физической памяти, что критично для современных систем с большим количеством RAM, таких как серверы или рабочие станции с интенсивным использованием памяти.
  - Поддержка виртуальной памяти, позволяющей программам работать с памятью, превышающей физическую, что улучшает многозадачность и производительность.
  - Уменьшение необходимости в файле подкачки при наличии достаточного объема RAM, что снижает нагрузку на диск и улучшает скорость работы.
- **Недостатки**:
  - Увеличение сложности управления таблицами страниц, что может приводить к дополнительным накладным расходам на обработку.

- Потенциальный рост накладных расходов на память для хранения таблиц страниц, особенно в системах с большим количеством процессов.
- Возможные задержки из-за пропусков в буфере трансляции (TLB), особенно в системах с большими адресными пространствами, что может снижать производительность.

Таблица 3

Сравнительная таблица архитектур

Архитектура	Уровни таблиц страниц	Максимальный объем памяти	Преимущества	Недостатки
x86 (32-bit)	2 (с PAE — 3)	4 ГБ (с PAE — 64 ГБ)	Совместимость, простота для малых систем	Ограничение на 4 ГБ на процесс, сложность с PAE
x86-64 (64-bit)	4	256 ТБ	Поддержка терабайт, высокая производительность	Высокие накладные расходы, сложность совместимости

31. Опишите основные функции Win32 для управления виртуальной памятью. Опишите переходы между различными списками страниц в Windows.

API Win32 предоставляет набор функций для управления виртуальной памятью в Windows. Эти функции позволяют выделять, освобождать, защищать и резервировать память в адресном пространстве процесса. Вот основные из них:

1. **VirtualAlloc**
   - **Назначение**: Выделяет или резервирует область виртуальной памяти в адресном пространстве процесса.
   - **Параметры**:
     - lpAddress: Указатель на начальный адрес (может быть NULL для автоматического выбора).
     - dwSize: Размер области в байтах.
     - flAllocationType: Тип выделения (MEM_COMMIT, MEM_RESERVE, MEM_TOP_DOWN, и т.д.).
     - flProtect: Атрибуты защиты памяти (PAGE_READWRITE, PAGE_EXECUTE, и т.д.).
   - **Пример использования**: Выделение памяти для буфера данных или стека.
2. **VirtualFree**
   - **Назначение**: Освобождает или декоммитирует (освобождает физическую память без удаления резервирования) область виртуальной памяти.
   - **Параметры**:
     - lpAddress: Начальный адрес области.
     - dwSize: Размер области (0 для полного освобождения).
     - dwFreeType: Тип освобождения (MEM_DECOMMIT, MEM_RELEASE).
   - **Пример использования**: Освобождение ранее выделенной памяти.
3. **VirtualProtect**
   - **Назначение**: Изменяет атрибуты защиты для области виртуальной памяти (например, чтение, запись, выполнение).
   - **Параметры**:
     - lpAddress: Начальный адрес области.
     - dwSize: Размер области.

        - flNewProtect: Новый атрибут защиты.
        - lpflOldProtect: Указатель для сохранения старого атрибута защиты.
    - **Пример использования**: Изменение прав доступа для предотвращения записи в код.
4. **VirtualQuery**
    - **Назначение**: Возвращает информацию о состоянии области виртуальной памяти.
    - **Параметры**:
        - lpAddress: Начальный адрес области.
        - lpBuffer: Указатель на структуру MEMORY_BASIC_INFORMATION, содержащую информацию (состояние, защита, размер).
        - dwLength: Размер буфера.
    - **Пример использования**: Проверка статуса памяти перед выделением или освобождением.
5. **VirtualLock**
    - **Назначение**: Блокирует область виртуальной памяти в физической памяти, предотвращая её выгрузку в файл подкачки.
    - **Параметры**:
        - lpAddress: Начальный адрес.
        - dwSize: Размер области.
    - **Пример использования**: Используется для критически важных данных, требующих быстрого доступа.
6. **VirtualUnlock**
    - **Назначение**: Разблокирует ранее заблокированную память, позволяя её выгрузку в файл подкачки.
    - **Параметры**: Аналогичны VirtualLock.
    - **Пример использования**: Снятие блокировки после завершения работы с данными.
7. **HeapAlloc**, **HeapFree**, **HeapCreate**, **HeapDestroy**
    - **Назначение**: Управление кучей, которая является частным случаем виртуальной памяти. Эти функции используются для выделения и освобождения памяти в куче процесса.
    - **Пример использования**: Динамическое выделение памяти для небольших объектов.
8. **MapViewOfFile**, **UnmapViewOfFile**
    - **Назначение**: Отображает файл в виртуальное адресное пространство процесса и освобождает отображение.
    - **Пример использования**: Работа с файлами подкачки или memory-mapped файлами.

Переходы между различными списками страниц в Windows

В Windows виртуальная память управляется с помощью страниц, которые находятся в различных списках, отражающих их состояние. Переходы между списками страниц происходят в зависимости от активности процесса, потребности в памяти и действий операционной системы. Основные списки страниц и переходы между ними описаны ниже:

Основные списки страниц

1. **Active List (Активный список)**:
    - Содержит страницы, которые активно используются процессами и находятся в физической памяти (рабочий набор процесса).
    - Страницы имеют атрибуты защиты (например, чтение/запись) и могут быть заблокированы (VirtualLock).
2. **Standby List (Список ожидания)**:
    - Содержит страницы, которые были вытеснены из активного списка, но всё ещё находятся в физической памяти и могут быть быстро восстановлены.
    - Эти страницы связаны с процессами, но временно не используются.

3. **Modified List (Список изменённых страниц)**:
   - Содержит страницы, которые были изменены в памяти и должны быть записаны в файл подкачки или на диск перед повторным использованием.
   - Страницы переходят сюда из активного списка, если они были изменены.
4. **Free List (Свободный список)**:
   - Содержит страницы, которые не содержат данных и готовы к немедленному использованию для новых выделений.
   - Страницы в этом списке очищаются (заполняются нулями) для безопасности.
5. **Zeroed List (Список обнулённых страниц)**:
   - Подмножество свободного списка, содержащее страницы, уже очищенные нулями и готовые к выделению.
   - Используется для повышения производительности при выделении памяти.
6. **Bad List (Список повреждённых страниц)**:
   - Содержит страницы, которые физически неисправны и не могут использоваться.

Переходы между списками

1. **Active List → Standby List**:
   - Когда страница в рабочем наборе процесса становится неактивной (например, из-за низкой частоты обращений), она перемещается в Standby List.
   - Это происходит, когда менеджер памяти решает сократить рабочий набор процесса для освобождения физической памяти.
2. **Active List → Modified List**:
   - Если страница в Active List была изменена (например, запись данных), она перемещается в Modified List, чтобы её содержимое могло быть записано в файл подкачки или на диск.
3. **Modified List → Standby List**:
   - После записи изменённых данных на диск или в файл подкачки страница перемещается в Standby List, где она может быть повторно использована без необходимости чтения с диска.
4. **Standby List → Active List**:
   - Если процесс обращается к странице, находящейся в Standby List, она быстро возвращается в Active List, так как её содержимое всё ещё в физической памяти.
5. **Standby List → Free List**:
   - Если страница в Standby List долго не используется, она может быть перемещена в Free List, чтобы освободить физическую память для других задач.
6. **Free List → Zeroed List**:
   - Страницы в Free List очищаются (заполняются нулями) фоновым процессом (Zero Page Thread) и перемещаются в Zeroed List, чтобы быть готовыми к немедленному выделению.
7. **Zeroed List → Active List**:
   - Когда процессу требуется новая страница памяти (например, через VirtualAlloc с MEM_COMMIT), страница берётся из Zeroed List и перемещается в Active List.
8. **Любой список → Bad List**:
   - Если физическая страница обнаруживается как неисправная (например, из-за аппаратной ошибки), она перемещается в Bad List и исключается из дальнейшего использования.

32. Опишите организацию главной таблицы файлов NTFS. Объясните значение полей структуры записи MFT для разреженного файла.

Файловая система NTFS, используемая в операционных системах Windows,

включает главную таблицу файлов (Master File Table, MFT), которая является центральным компонентом для хранения метаданных. MFT представляет собой файл, расположенный на диске, и содержит записи для каждого файла и директории на томе. Каждая запись MFT обычно имеет размер 1024 байта и включает различные атрибуты, описывающие свойства файла, такие как имя, даты создания и изменения, права доступа и данные.

Разреженные файлы — это особый тип файлов в NTFS, которые позволяют создавать очень большие файлы, но при этом занимать на диске только столько места, сколько необходимо для хранения ненулевых данных. Это достигается за счет того, что диапазоны данных, состоящие из нулей, не записываются на диск, а вместо этого отмечаются как "разреженные" (sparse). При чтении таких диапазонов система возвращает нули, не обращаясь к диску, что делает разреженные файлы эффективными для хранения, например, больших матриц данных или изображений с большими областями нулей.

Организация MFT в NTFS

MFT организована как последовательность фиксированных записей, каждая из которых представляет файл или директорию. MFT начинается с набора зарезервированных записей для системных файлов, таких как сам MFT ($MFT), зеркало MFT ($MFTMirr), файл логов ($LogFile) и другие. Следующие записи используются для пользовательских файлов и директорий.

Каждая запись MFT состоит из следующих основных частей:

- **Заголовок записи файла (File Record Header)**: Содержит информацию о самой записи, включая размер, смещение последовательности обновления и флаги, указывающие на состояние файла (например, удален или активен).
- **Список атрибутов файла (File Attribute List)**: Включает набор атрибутов, таких как:
  - **$STANDARD_INFORMATION**: Хранит стандартные метаданные, включая даты создания, последнего изменения, последнего доступа и удаления, а также флаги, указывающие, является ли файл каталогом или обычным файлом.
  - **$FILE_NAME**: Содержит имя файла или директории, включая путь и другие связанные данные, такие как короткое имя для совместимости.
  - **$SECURITY_DESCRIPTOR**: Хранит информацию о правах доступа и безопасности файла.
  - **$DATA**: Ключевой атрибут, который содержит данные файла или указывает на их расположение на диске через список run-ов (последовательности кластеров).

Эта структура делает MFT простой для парсинга и эффективной для доступа к метаданным. MFT выделяет определенное количество пространства для каждой записи файла, и атрибуты записываются в это пространство. Маленькие файлы и директории (обычно 512 байт или меньше) могут полностью содержаться в записи MFT, что ускоряет доступ к файлу.

Разреженные файлы и их представление в MFT

Разреженные файлы оптимизируют хранение, выделяя дисковое пространство только для ненулевых данных. В MFT для таких файлов атрибут $DATA содержит список "run-ов", которые отображают виртуальные кластерные номера (VCN, Virtual Cluster Number) на логические кластерные номера (LCN, Logical Cluster Number). Каждый run указывает диапазон VCN и соответствующий диапазон LCN, где данные фактически хранятся на диске.

Для разреженных файлов в списке run-ов могут присутствовать диапазоны VCN, которые не отображаются на LCN. Такие диапазоны считаются разреженными и не занимают места на диске. При чтении таких диапазонов система возвращает нули, что делает доступ к файлу прозрачным для приложений. Например, файл размером 1 ГБ, содержащий только 1 МБ ненулевых данных, будет иметь остальные 999 МБ как разреженные диапазоны в атрибуте $DATA.

Кроме того, флаг, указывающий на разреженность файла, находится в атрибуте

$STANDARD_INFORMATION. В поле флагов (offset 32-35) бит 0x0200 указывает, что файл является разреженным. Это позволяет системе идентифицировать файл как разреженный и соответствующим образом управлять его данными.

Структура run-ов для разреженных файлов

Runlist для атрибута $DATA разреженного файла включает специальные записи для разреженных участков. Каждый элемент runlist состоит из заголовка, указывающего размер полей длины и смещения, за которыми следуют значения длины и смещения. Для разреженных run-ов размер поля смещения равен 0, что означает, что нет соответствующего LCN, и длина указывает количество кластеров, которые являются разреженными и не занимают места на диске.

Например, рассмотрим файл, где первые 5 кластеров содержат данные, следующие 3 кластера — нули (разреженные), и последние 4 кластера — данные. Runlist может выглядеть так:

- Run 1: VCN 0-4 → LCN 100-104 (данные записаны на диске).
- Run 2: VCN 5-7 → нет LCN (разреженный, размер смещения = 0, длина = 3).
- Run 3: VCN 8-11 → LCN 200-203 (данные записаны на диске).

При чтении диапазона VCN 5-7 система вернет нули, не обращаясь к диску, что экономит ресурсы.

33. Раскройте сущность вопросов, касающихся реализации безопасности в Windows. Приведите основные функции безопасности в Win32.

Модель безопасности Windows основана на концепции защищаемых объектов, таких как процессы, потоки, события, файлы, каталоги и устройства. Каждый объект имеет дескриптор безопасности, который включает идентификаторы безопасности (SID) владельца и группы, а также списки контроля доступа (ACL), определяющие права доступа для пользователей и групп.

Типичные вопросы, возникающие при реализации безопасности, включают:

- Как настроить ACL для защиты файлов и ключей реестра?
- Как аутентифицировать пользователей и управлять их контекстом безопасности?
- Как шифровать данные для обеспечения конфиденциальности?
- Как настроить аудит событий безопасности для мониторинга?

Эти вопросы важны как для разработчиков, создающих приложения с учетом безопасности, так и для системных администраторов, управляющих доступом и политиками. Например, разработчики могут задаваться вопросом, как ограничить доступ к ресурсам, используя функции Win32, такие как GetSecurityInfo или SetSecurityInfo. Администраторы, в свою очередь, могут интересоваться настройкой групповых политик или управлением учетными записями.

Модель безопасности Windows также включает концепции, такие как токены доступа, содержащие SID пользователя, группы и привилегии, а также системные привилегии для определенных операций. Например, драйверы должны обеспечивать безопасность своих устройств, устанавливая соответствующие дескрипторы безопасности, часто указываемые в INF-файлах с использованием языка SDDL (Security Descriptor Definition Language).

Основные функции безопасности в Win32

Win32 API предоставляет обширный набор функций для реализации различных аспектов безопасности. Эти функции можно разделить на несколько категорий, каждая из которых решает конкретные задачи. Ниже представлена таблица с примерами функций и их описанием:

Таблица 4

Основные функции безопасности в Win32

Категория	Примеры функций	Описание
Аутентификация	LogonUser, ImpersonateLoggedOnUser, InitializeSecurityContext, AcceptSecurityContext	Проверка учетных данных, управление контекстом безопасности, поддержка протоколов, таких как Kerberos и NTLM.
Управление доступом	GetSecurityInfo, SetSecurityInfo, AddAce, AccessCheck, AddAccessAllowedAce	Работа со списками контроля доступа (ACL), проверка прав доступа, настройка дескрипторов безопасности.
Криптография	CryptAcquireContext, CryptEncrypt, CryptDecrypt, CryptHashData	Шифрование, дешифрование, хеширование и цифровые подписи с использованием CryptoAPI и CNG.
Управление сертификатами	CertOpenStore, CertFindCertificateInStore, CertVerifyCertificateChainPolicy	Работа с цифровыми сертификатами, их проверка и хранение.
Безопасная коммуникация	InitializeSecurityContext, AcceptSecurityContext (SSPI)	Установление защищенных соединений через SSL/TLS и другие протоколы.

Эти функции охватывают широкий спектр задач, от проверки подлинности пользователей до защиты данных и управления доступом. Например, LogonUser позволяет аутентифицировать пользователя, а AccessCheck проверяет, имеет ли пользователь необходимые права для доступа к объекту.

Кроме того, существуют функции для управления токенами, такими как CheckTokenMembership, для проверки членства в группах, и функции для работы с привилегиями, такими как AdjustTokenPrivileges. Также доступны функции для аудита безопасности, такие как AddAuditAccessAce, и управления TPM (Trusted Platform Module) через функции, такие как TBS_InitializeContext.

34. Объясните значение полей структуры записи MFT для каталога.

Объясните на примере принцип прозрачного сжатия файлов в Windows.

Первые 16 записей MFT резервируются для файлов метаданных NTFS. Каждая из этих записей описывает нормальный файл, который имеет атрибуты и блоки данных.

- **Первая запись (0)** описывает сам файл MFT. В частности, в ней говорится, где находятся блоки файла MFT (чтобы система могла найти файл MFT).
- **Запись 1** является дубликатом начала файла MFT. Эта информация настолько ценная, что наличие второй копии может быть просто критическим (в том случае, если один из первых блоков MFT испортится).
- **Запись 2** — файл журнала. Когда в файловой системе делаются структурные изменения, то такое действие журналируется здесь до его выполнения (чтобы повысить вероятность корректного восстановления в случае сбоя во время операции — например такого, как отказ системы).
- **Запись 3** содержит информацию о томе (такую, как его размер, метка и версия).
- **Запись 4** – содержит информацию о системном файле $AttrDef, который определяет

все возможные типы атрибутов, используемых в структуре MFT.

- **Запись 5** – описывает корневой каталог, который сам является файлом и может расти до произвольного размера.
- **Запись 6** – хранит атрибуты и дисковые адреса файла с битовым массивом, который отслеживает свободное пространство тома.
- **Запись 7** – указывает на файл начального загрузчика.
- **Запись 8** – используется для того, чтобы связать вместе все плохие блоки (чтобы обеспечить невозможность их использования для файлов).
- **Запись 9** – содержит информацию безопасности.
- **Запись 10** – используется для установления соответствия регистра.
- **Запись 11** – это каталог, содержащий различные файлы для таких вещей, как дисковые квоты, идентификаторы объектов, точки повторной обработки и т. д.
- **Записи 12-15 -** зарезервированы для использования в будущем.

Принцип работы прозрачного сжатия на примере:

Допустим, у вас есть текстовый файл report.txt размером **1 МБ**. Вы включаете сжатие для этого файла через свойства проводника:

1. **Сжатие включается**: Windows использует встроенный алгоритм (например, LZNT1) и сжимает содержимое файла, скажем, до **400 КБ**. В MFT (главной таблице файлов) файл помечается как "сжатый". На физическом уровне на диске сохраняется уже сжатая версия данных.
2. **Открытие файла**: Когда вы или приложение открываете report.txt, ОС **прозрачно распаковывает** данные в оперативной памяти. Пользователь этого не замечает — файл открывается как обычно, и он снова выглядит как 1 МБ.
3. **Изменение и сохранение**: Когда вы сохраняете изменения, Windows снова сжимает файл в фоне перед записью на диск.

35. Объясните принцип работы системы шифрования файлов в Windows. Охарактеризуйте отличия реализации технологии шифрования данных BitLocker Drive Encryption.

Когда пользователь включает шифрование файла, Windows создаёт уникальный случайный ключ (128/192/256 бит в зависимости от версии ОС и используемого алгоритма) и использует его для поблочного шифрования файла с помощью симметричного алгоритма. Ключ хранится так же на диске, и чтобы сохранить этот ключ в безопасности используется шифрование с помощью открытого ключа пользователя. При открытии файла система извлекает зашифрованный симметричный ключ, расшифровывает его с помощью закрытого ключа пользователя, и только затем дешифрует содержимое файла.

Отличия технологий:

Основное отличие в том, **на каком уровне работает технология**:

- BitLocker работает **на уровне тома** (диска) и обеспечивает шифрование **всех данных на устройстве**, включая системную область.
- EFS — работает **на уровне файловой системы NTFS**, позволяя шифровать только нужные файлы или каталоги.

В отличие от EFS, BitLocker защищает данные ещё до запуска операционной системы, и это делает невозможным даже загрузку ОС без правильного ключа.

BitLocker не зависит от конкретных пользователей — защищает весь диск целиком, а не отдельные профили. Может работать даже на неактивных томах, в то время как EFS требует, чтобы пользователь вошёл в систему.

В итоге, BitLocker — это инструмент для глобальной защиты устройства, а EFS — для избирательного шифрования данных. BitLocker применяется в случаях, когда нужно гарантировать конфиденциальность всего содержимого накопителя, даже при его физическом изъятии.

36. Сформулируйте критерии и приведите методы оценки эффективности конструкторских решений, реализованных в системе управления памятью в Windows семейства NT.

1. Эффективность выделения блоков
Критерий: минимизация числа разрозненных фрагментов при записи больших или разбросанных потоков.
Методы оценки:
- Смоделировать запись «потока» разных размеров и степеней фрагментации (от 1 до n участков) и проанализировать количество записей MFT, задействованных для описания его кластеров.
- Подсчитать, сколько пар (дисковый адрес + длина) понадобилось на один и тот же объём данных при непрерывном и при разреженном размещении.

2. Внешняя и внутренняя фрагментация
Критерий: степень неиспользуемого пространства в рамках выделенных кластеров и «дыр» между ними.
Методы оценки:
- Измерить относительный объём «пустых» байт внутри сжатых и несжатых областей (учитывая, что пары записей MFT могут описываться сжатым форматом — 4…16 байт вместо 16 байт).
- Проследить число свободных «кусочков» (MEMORY_BASIC_INFORMATION-аналог в файловом мире — битовый массив свободных кластеров) и оценить, какой наибольший непрерывный участок остаётся доступным после серии операций создания/удаления.

3. Масштабируемость MFT
Критерий: способность MFT расти (до 2^{48} записей) и обслуживать большое число файлов без значительного ухудшения производительности.
Методы оценки:
- Формировать каталоги и файлы так, чтобы требовалось задействовать более одной записи MFT (base record + записи расширения), и затем измерить «глубину» цепочки расширений и время обхода этой цепочки при поиске/открытии файла.
- Проверить время доступа к файлу, когда MFT размещается не в начале тома, а «по всей поверхности» (учёт расположения метаданных как в записи 0 и 1).

4. Надёжность через журналирование
Критерий: скорость восстановления целостности после имитации сбоя при структурных изменениях (создание/удаление каталогов, изменение атрибутов).
Методы оценки:
- Сделать серию операций изменения структуры и форсированно прервать питание, затем замерить время и объём работы по «откату» или «доделке» через файл-журнал (запись 2).
- Оценить, сколько записей из журнала применяется до достижения консистентного состояния.

5. Сжатие и его прозрачность

Критерий: компромисс между степенью сжатия и накладными расходами на алгоритм LZNT1 по 16-блокам (или по 32…631 блокам).

Методы оценки:

- Для заданного потока данных разбить его на фрагменты по 16 логических блоков и измерить, в каком проценте случаев они сжимаются до ≤ 15 блоков (и записываются как единый экстент).
- Измерить общее соотношение сжатого к исходному объёму и время упаковки/распаковки при последовательном чтении.

6. Безопасность метаданных (атрибутов и ключей)

Критерий: надёжность хранения критичных файлов (MFT — запись 0, дублирующая запись 1, $AttrDef — запись 4) и зашифрованных ключей (EFS).

Методы оценки:

- Проверить устойчивость к повреждению первой записи MFT: симулировать дефектные сектора в записи 0 и убедиться, что система корректно «перескакивает» на запись 1.
- Произвести шифрование EFS-файла и убедиться, что его симметричный ключ хранится только в зашифрованном виде рядом с файлом (метаданные записи), а расшифровка возможна лишь при наличии закрытого ключа пользователя.

37. Сформулируйте критерии и проведите сравнительный анализ эффективности алгоритмов планирования распределения ЦП, используемых в UNIX и Linux.

1. Адаптивность к интерактивным (I/O-ограниченным) задачам

- **UNIX**:
  - Процессы, вернувшиеся из состояния ожидания (I/O, семафор и т. д.), получают **отрицательную “базовую” приоритетную метку** и попадают в высокоприоритетную очередь.
  - Приоритет динамически растёт с накоплением CPU_usage и снижается по экспоненциальному закону (раз в тик: (старое + тики)/2), что даёт быстрый отклик “спящим” процессам.
- **Linux**:
  - После блокировки на I/O поток остаётся в своей очереди с остатком кванта; при пересчёте квант увеличивается по формуле new_quantum = (old_quantum/2) + priority, что даёт I/O-ограниченным потокам **больший следующий квант** и, соответственно, лучший доступ к CPU.

Вывод: обе системы делают упор на быстрое возвращение I/O-ограниченных задач в исполнение, но UNIX делает это через мгновенную «пересадку» в высокую очередь, а Linux — через наращивание квантов времени.

2. Структура очередей и приоритетов

- **UNIX**:
  - Многоканальная (multilevel) схема: каждая очередь соответствует диапазону приоритетов; ядро (системные вызовы) имеет отрицательные приоритеты — высший уровень.
  - Пересчёт приоритета происходит при каждом тике, и номер очереди = (priority / константа).
- **Linux**:
  - Три **класса** потоков:
    1. Реального времени FIFO (наивысший приоритет, без таймерных прерываний)

2. Реального времени RR (циклическая очередь с квантами)
3. Time-sharing (обычные потоки)
   - Для любых потоков вычисляется **“добродетель”** (goodness):
     - Real-time: 1000 + priority
     - Timesharing (quantum > 0): quantum + priority
     - Timesharing (quantum = 0): 0
   - Каждый тик квант уменьшается, а когда все кванты “исчерпаны”, для всех потоков пересчитываются кванты как (old_quantum/2)+priority.

Вывод: UNIX опирается на набор фиксированных очередей с постоянным диапазоном приоритетов и непрерывным обновлением, а Linux — на два реального-временных класса и один time-sharing, при этом «глобальное» решение о следующем потоке принимается через максимизацию значения goodness.

3. Гарантии реального времени и изоляция

- **UNIX**:
  - Нет отдельного «real-time» класса: всё сводится к отрицательным приоритетам для I/O и системных вызовов, но жёстких гарантий по срокам нет.
- **Linux**:
  - Два чётко выделенных real-time класса (FIFO и RR) с приоритетами выше любых time-sharing потоков.
  - Хотя реальные дедлайны (hard RT) не обеспечиваются, эти потоки **не могут быть вытеснены** обычными, что даёт более предсказуемое поведение для soft RT-задач.

Вывод: Linux вводит формальную поддержку soft real-time через отдельные классы, тогда как в UNIX всё укладывается в единый многоуровневый приоритет без явного отделения RT.

4. Накладные расходы и сложность перерасчётов

- **UNIX**:
  - Приоритет каждого процесса обновляется **каждый таймерный тик** (сложение и сдвиг/деление на 2), плюс распределение по очередям.
  - Требует управления множеством очередей и быстрых переключений между ними.
- **Linux**:
  - Goodness вычисляется **лишь при выборе очередного потока**; квант обновляется пачками, когда все кванты равны нулю.
  - Меньше перманентных перерасчётов приоритета, но есть периодическая фаза “регенерации” квантов.

Вывод: UNIX платит за непрерывное взвешивание CPU_usage, Linux — за периодический пересчёт квантов и эффективность вычисления goodness только раз за квант.

5. Предотвращение голодания и справедливость

- **UNIX**:
  - Адаптивное старение через деление CPU_usage гарантирует, что «зажравшийся» CPU-бёрн практически со временем потеряет приоритет.
  - I/O-процессы получают «быстрый билет» в высокую очередь.
- **Linux**:
  - Добродетель (goodness) сводит квант и приоритет в единое значение: даже если поток высокоприоритетный, квант постепенно истощается, но при пересчёте ему вновь выделят хотя бы priority тиков.
  - I/O-ограниченные получают бонус к квантам, а потоки, активно нагружающие CPU, постепенно сбрасываются к своему приоритету.

Вывод: обе системы сбалансированы по принципу «награда для I/O-ограниченных — штраф для CPU-ограниченных», однако механизмы старения и перераспределения ресурсов различаются по частоте и «гранулярности».

Общий вывод

- **UNIX** делает ставку на **многоканальные очереди** и **постоянное обновление** приоритета на основе CPU_usage и базовой метки I/O, что даёт быстрый отклик интерактивным задачам, но требует частых перерасчётов.
- **Linux** вводит чёткое **разделение на real-time и time-sharing**, рассчитывает «добродетель» лишь при выборе следующего потока и **пересчитывает квант** пакетно, что упрощает логику и одновременно сохраняет адаптивность к I/O-нагрузкам.

38. Предложите варианты монтирования удаленных каталогов в NFS и обоснуйте их на основе сравнения.

1. Статическое монтирование при загрузке (/etc/exports + /etc/rc или /etc/fstab)

Описание:

На сервере каталоги экспортируются через файл /etc/exports. На клиенте в скрипте загрузки (например, /etc/rc) или в /etc/fstab прописываются соответствующие записи вида
server1:/exported/dir /mnt/dir nfs defaults 0 0
и они автоматически примонтируются при старте ОС.

Плюсы:

- Очень простой и предсказуемый механизм.
- Монтирование происходит до входа пользователей в систему — все ресурсы сразу доступны.

Минусы:

- Если один из серверов недоступен (выключен, по сети отказ), загрузка клиента зависнет или завершится ошибкой монтирования.
- Нет отказоустойчивости и балансировки нагрузки — всегда пытаемся к одному и тому же серверу.

2. Автомонтирование по требованию (autofs)

Описание:

Вместо того чтобы монтировать всё сразу, в таблице автомонтировщика (/etc/auto.master и т. п.) задаются каталоги, за которыми могут стоять удалённые ресурсы. Фактический NFS-маунт выполняется **только при первом обращении** к этому пути.

Плюсы:

- Клиент не тратит время и ресурсы на монтирование «ненужных» каталогов.
- Сокращаются задержки при старте системы (монтирование лишь по факту обращения).

Минусы:

- Первое обращение к ресурсу может «задёргаться» дольше из-за фактического монтирования.
- Требуется настроить и поддерживать сервис automount.

3. Параллельное автомонтирование нескольких серверов (авторотация / отказоустойчивость)

Описание:

При использовании automount (или специально скриптов), для одного и того же локального каталога задаётся **несколько удалённых серверов** в качестве кандидатов. Клиент одновременно посылает запросы на все адреса и **примонтирует каталог** того сервера, который ответит **первым**.

Плюсы:

- **Устойчивость к отказам**: если один сервер недоступен, монтирование пройдёт автоматически на другом.
- **Балансировка нагрузки**: первым ответит наименее загруженный или менее загруженный сетью сервер, что снижает задержки.

Минусы:

- Сложнее конфигурация automount-карт (несколько записей для одного маунт-поинта).
- Требуется гарантия синхронизации данных между серверами, иначе разные клиенты будут видеть разные версии файлов.

Обоснование выбора:
- Если важна простота и сеть надёжна — достаточно статического монтирования.
- Если нужна оптимизация загрузки и монтирование только при надобности — выбирают automount.
- Если требуется максимум отказоустойчивости и минимальные задержки при первом доступе — используют параллельное автомонтирование нескольких серверов, где клиент сам «выбирает» лучший ресурс.

39. Предложите вариант использования команды su и сформулируйте критерии запуска.

Вариант использования:

```
su -c 'mount -a' root
```

Здесь:
1. su -c '…' запускает в контексте суперпользователя (root) ровно одну команду — mount -a,
2. Параметр - (иногда пишут -l или --login) гарантирует, что при выполнении команды загрузятся все переменные окружения и профиль root (важно для корректной работы скриптов в /etc/rc, автомонтировщика autofs и т. п.).
3. После завершения команды вы автоматически возвращаетесь в свой обычный пользовательский сеанс.

Критерии запуска su
1. **Доступ к «корневым» ресурсам файловой системы**
   - Монтирование/отключение NFS-каталогов (mount, umount)
   - Изменение экспортов в /etc/exports или прав в /etc/fstab
   - Управление демонами autofs, правка их конфигов
2. **Изменение системных настроек и служб**
   - Изменение расписания (cron, at) или управление приоритетами задач (nice, renice)
   - Запуск или остановка системных демонов (например, NFS-сервера, demount)
3. **Работа с защищёнными файлами и метаданными**
   - Прямой доступ к MFT-аналогам, системным разделам, логам (в контексте NTFS-подобных схем, если бы был порт UNIX)
   - Выполнение операций, требующих прав на запись в /boot, /etc, /var
4. **Корректная среда выполнения**
   - Когда команда или скрипт рассчитывает на окружение root (пути /root/.profile, особые переменные)
   - Чтобы избежать «засорения» вашего пользовательского $PATH или алиасов и гарантировать одинаковое поведение
5. **Минимизация рисков**
   - Использовать su -c вместо полноценного su (или su - root), чтобы **сократить время под правами суперпользователя** и снизить шанс случайного изменения чего-либо критичного
   - Фиксировать команды с su в скриптах и логировать их выход, чтобы при сбое было понятно, на каком этапе

40. Предложите варианты команд для добавления, удаления и модификации информации о пользователях системы в ОС FreeBSD и обоснуйте своё предложение.

Варианты:

1. **Утилита** pw**:**

Добавление:

pw useradd alice -c "Alice Ivanova" -d /home/alice -m -s /usr/local/bin/bash -G wheel

- -c — комментарий (полное имя)
- -d — домашний каталог; -m автоматически создаёт его
- -s — shell; -G — дополнительные группы (например, wheel для sudo)

Удаление:

pw userdel alice -r

- -r – удаляет домашний каталог и почтовый спул

Модификации:

pw usermod alice \

- -l alicia \ – изменить логин
- -s /bin/tcsh \ – сменить shell
- -G staff,wheel – изменить список дополнительных групп

Почему pw**?**

- Это родная утилита FreeBSD, она умеет обновлять все необходимые файлы (/etc/master.passwd, /etc/group, /etc/pwd.db)
- Подходит и для скриптов, и для администрирования «вручную», так как имеет понятный синтаксис

2. **Встроенный мастер** adduser

Добавление:

adduser

При старте мастера вы шаг за шагом задаёте: имя учётки, UID, группу, домашний каталог, shell, пароль и т. п.

Удаление и модификация:

Для удаления после adduser всё равно придётся использовать pw userdel, а для модификации — pw usermod или vipw.

Когда использовать adduser?

- При ручном создании разовых учётных записей
- Когда нужно минимизировать опечатки в синтаксисе команд (мастер задаёт вопросы)

3. **Прямое редактирование и утилиты** vipw/vigr

Добавление/удаление/модификация:

- Запускаем:
  vipw
- Отредактируйте файл /etc/master.passwd (или /etc/group) вручную:
  alice:*:1001:1001:Alice Ivanova:/home/alice:/usr/local/bin/bash
- По выходу система автоматически перекомпилирует базы /etc/pwd.db и /etc/spwd.db.

Когда vipw оправдан?

- Нужна тонкая ручная правка (нестандартный формат, правка сразу нескольких записей)

- Сценарии аварийного восстановления, когда pw “ломается” или недоступен

41. Предложите варианты разработки программных продуктов в ОС FreeBSD без установки дополнительного ПО, покажите плюсы и минусы разных подходов.

Варианты:

1. Ручная компиляция через cc

```
cc -Wall -O2 main.c util.c -o myapp
```

Плюсы:

- Максимально просто: нет Makefile, всё видно в одной строке.
- Нулевая зависимость: используется только базовый компилятор.

Минусы:

- Ручное управление зависимостями: при любом добавлении нового .c или заголовка придётся менять команду.
- Хрупкость: легко забыть флаг, дефайн или путь к инклуду.

2. Простейший Makefile с bsd.prog.mk

В FreeBSD в /usr/share/mk лежит набор include-файлов, которые автоматически задают флаги, пути, правила линковки и установки.

```
# Makefile
PROG=   myapp
SRCS=   main.c util.c
MAN=    myapp.1

.include <bsd.prog.mk>
```

Плюсы:

- Минимум кода: все правила «из коробки».
- Консистентность: те же флаги (CC, CFLAGS, LDFLAGS) что у всей системы.
- Поддержка make install и генерации man-страниц.

Минусы:

- Нужно понять, как устроен bsd-mk, — чуть более крутая входная кривая, чем в «чистой» GNU-make.

3. Рекурсивный (иерархический) make

Разбить проект на модули:

```
src/
core/Makefile     # собирает libcore.a
net/ Makefile     # собирает libnet.a
app/ Makefile     # собирает myapp, линкуя с libcore, libnet
Makefile          # .PHONY: all → субдиректории
```

Плюсы:

- Масштабируемость: удобно расти до десятков папок.
- Локальные правила: каждый модуль сам знает, что и как компилировать.

Минусы:

- Сложность зависимостей: особенно если одни модули часто меняют интерфейс.
- «Опасность» рекурсивного make: общие цели и переменные могут «не дойти» до субдеревьев.

4. Скриптовая обвязка на sh/ksh **+ стандартные утилиты**

Можно обойтись без make, написав в корне:

```
#!/bin/sh
set -e
SRC="main.c util.c"
OBJ=""
for f in $SRC; do
  o=${f%.c}.o
  cc -c -Wall -O2 $f -o $o
  OBJ="$OBJ $o"
done
cc $OBJ -o myapp
```

Плюсы:

- Полный контроль над порядком действий.
- Нет magic — всё видно в одном скрипте.

Минусы:

- Самописный «build system» редко оказывается надёжнее проверенного make.
- Труднее добавлять новые цели (тесты, docs, install).

42. Предложите варианты использования специальных битов безопасности “setuid”, “setgid” и “sticky”.

1. Бит setuid:
   - Применение: Используется для временного предоставления процессу прав владельца программы (обычно суперпользователя, root) при её выполнении. Это позволяет обычным пользователям выполнять действия, требующие повышенных привилегий, без предоставления полного доступа к системе.
   - Пример: Программа passwd, которая позволяет пользователю менять свой пароль. Эта программа имеет владельца root и установлен бит setuid. Благодаря этому она может изменять файл паролей (/etc/passwd), но только для пароля вызвавшего её пользователя, не затрагивая остальное содержимое файла. Это обеспечивает безопасность, так как пользователь не получает прямого доступа к файлу паролей.
2. Бит setgid:
   - Применение: Аналогичен setuid, но временно предоставляет процессу права группы, связанной с файлом программы. Позволяет выполнять действия от имени группы владельца файла.
   - Пример: бит setgid используется редко, но он может применяться для программ, которые должны работать с ресурсами, доступными определённой группе пользователей. Например, программа, работающая с файлами, доступными только определённой группе, может использовать setgid для предоставления временного доступа к этим файлам.
3. Бит sticky:
   - Применение: sticky-бит обычно используется для каталогов (например, /tmp), чтобы ограничить удаление файлов в каталоге. Только владелец файла или суперпользователь может удалить файл, даже если другие пользователи имеют права на запись в каталог.
   - Пример: Каталог /tmp, где множество пользователей создают временные файлы. Установка sticky-бита (режим t, например, rwxrwxrwt) гарантирует, что пользователи не смогут удалять чужие файлы, несмотря на общие права на запись.

43. Приведите алгоритм установки X.

Версией X для FreeBSD по умолчанию является Xorg. Алгоритм установки следующий:

1. Подготовка к установке:
   - Убедитесь, что у вас есть не менее 4 ГБ свободного дискового пространства для полной установки Xorg.
   - Проверьте наличие необходимых зависимостей и совместимость оборудования.
2. Установка Xorg:
   - Вариант 1: Установка с помощью менеджера пакетов:
     - Выполните команду для загрузки и установки последней версии Xorg:
       ```
       # pkg install xorg
       ```
       Эта команда автоматически установит полный дистрибутив X11, включая сервер, клиенты, шрифты и другие компоненты.
   - Вариант 2: Установка из коллекции портов:
     - Перейдите в каталог порта Xorg:
       ```
       # cd /usr/ports/x11/xorg
       ```
     - Выполните сборку и установку:
       ```
       # make install clean
       ```
     - После установки убедитесь, что Xorg установлен корректно, проверив наличие файлов в /usr/local/bin.

44. Приведите алгоритм, как сконфигурировать X.

1. **Сбор информации об оборудовании**:
   - Соберите данные о мониторе (диапазоны частот горизонтальной и вертикальной синхронизации, указанные в документации или на сайте производителя).
   - Определите набор микросхем графического адаптера и объём видеопамяти, чтобы выбрать подходящий драйвер и настройки разрешения.
2. **Автоматическая настройка (если применимо)**:
   - Запустите Xorg без конфигурационного файла для проверки автоматической настройки:
     ```
     # startx
     ```
   - Для версий Xorg 7.4 и выше включите сервисы HAL и D-Bus, добавив в /etc/rc.conf:
     ```
     hald_enable="YES"
     dbus_enable="YES"
     ```
   - Перезапустите систему, чтобы применить настройки, и запустите Xorg.
3. **Создание начального конфигурационного файла**:
   - Если автоматическая настройка не работает или требуется специфическая конфигурация, создайте начальный файл xorg.conf.new:
     ```
     # Xorg -configure
     ```
     Это создаст файл /root/xorg.conf.new, содержащий настройки для обнаруженного оборудования.
4. **Тестирование конфигурации**:
   - Протестируйте созданный файл конфигурации:
     ```
     # Xorg -config xorg.conf.new
     ```
   - Для версий Xorg 7.4 и выше используйте опцию -retro, если видите только чёрный экран:
     ```
     # Xorg -config xorg.conf.new -retro
     ```
   - Если появляется чёрно-белая сетка и курсор в виде буквы X, настройка успешна. Завершите тестирование комбинацией Ctrl+Alt+Backspace.
   - Для активации Ctrl+Alt+Backspace в версиях Xorg 7.4 и выше:
     - Выполните в X-терминале:
       ```
       % setxkbmap -option terminate:ctrl_alt_bksp
       ```
     - Или создайте файл /usr/local/etc/hal/fdi/policy/x11-input.fdi с содержимым:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<deviceinfo version="0.2">
  <device>
    <match key="info.capabilities" contains="input.keyboard">
      <merge                        key="input.x11_options.XkbOptions"
type="string">terminate:ctrl_alt_bksp</merge>
    </match>
  </device>
</deviceinfo>
```

- Перезагрузите систему для применения настроек HAL.

5. **Тонкая настройка конфигурационного файла**:
   - Откройте /root/xorg.conf.new в текстовом редакторе и настройте параметры:
     - **Настройка монитора**:
       - В секции Monitor укажите частоты синхронизации:
```
Section "Monitor"
    Identifier "Monitor0"
    VendorName "Monitor Vendor"
    ModelName "Monitor Model"
    HorizSync 30-107
    VertRefresh 48-120
    Option "DPMS"
EndSection
```
       - Включите DPMS (если поддерживается монитором) для энергосбережения.
     - **Настройка разрешения и глубины цвета**:
       - В секции Screen задайте разрешение и глубину цвета по умолчанию:
```
Section "Screen"
    Identifier "Screen0"
    Device "Card0"
    Monitor "Monitor0"
    DefaultDepth 24
    SubSection "Display"
        Viewport 0 0
        Depth 24
        Modes "1024x768"
    EndSubSection
EndSection
```
     - **Настройка широкоэкранного режима (при необходимости)**:
       - Добавьте нужное разрешение (например, 1680x1050) в секцию Screen:
```
Modes "1680x1050"
```
       - Если требуется, создайте строку ModeLine на основе данных из лога /var/log/Xorg.0.log (например, EDID-данные):
```
Section "Monitor"
    Identifier "Monitor1"
    VendorName "Bigname"
    ModelName "BestModel"
    ModeLine "1680x1050" 146.2 1680 1784 1960 2240 1050 1053 1059
1089
    Option "DPMS"
EndSection
```
     - **Настройка мыши и клавиатуры**:
       - Если автоматическое определение устройств ввода (через HAL) некорректно, отключите его, добавив в секцию ServerLayout или ServerFlags:
```
Option "AutoAddDevices" "false"
```
       - Настройте раскладку клавиатуры, создав файл /usr/local/etc/hal/fdi/policy/x11-input.fdi:
```
<?xml version="1.0" encoding="iso-8859-1"?>
<deviceinfo version="0.2">
  <device>
    <match key="info.capabilities" contains="input.keyboard">
```

```
        <merge                          key="input.x11_options.XkbModel"
type="string">pc102</merge>
        <merge                        key="input.x11_options.XkbLayout"
type="string">fr</merge>
      </match>
    </device>
</deviceinfo>
```

- Или выполните команду в X-терминале:
  ```
  % setxkbmap -model pc102 -layout fr
  ```

6. **Установка конфигурационного файла**:
   - Скопируйте протестированный файл конфигурации в общедоступное место:
     ```
     # cp xorg.conf.new /etc/X11/xorg.conf
     ```
     или
     ```
     # cp xorg.conf.new /usr/local/etc/X11/xorg.conf
     ```
7. **Запуск Xorg**:
   - Запустите Xorg с помощью команды:
     ```
     # startx
     ```
   - Или настройте менеджер дисплеев (например, XDM) для автоматического запуска Xorg.
8. **Диагностика проблем**:
   - Проверьте лог-файлы /var/log/Xorg.0.log (или Xorg.1.log, Xorg.2.log и т.д.) для выявления ошибок.
   - При необходимости скорректируйте настройки в xorg.conf на основе информации из логов.

45. Приведите алгоритм установки дополнительных шрифтов в X.

• **Установка шрифтов Type1**:

1. Выберите коллекцию шрифтов, например, urwfonts или freefonts, из коллекции портов FreeBSD.
2. Выполните команды для установки:
   ```
   # cd /usr/ports/x11-fonts/urwfonts
   # make install clean
   ```
   Аналогично для freefonts:
   ```
   # cd /usr/ports/x11-fonts/freefonts
   # make install clean
   ```
3. Добавьте путь к шрифтам в конфигурацию X-сервера:
   - Отредактируйте файл /etc/X11/xorg.conf, добавив строку в секцию Files:
     ```
     FontPath "/usr/local/lib/X11/fonts/URW/"
     ```
   - Или выполните команду из X-терминала:
     ```
     # xset fp+ /usr/local/lib/X11/fonts/URW
     # xset fp rehash
     ```

     (Эти команды временные и теряются после завершения сеанса X, если не добавить их в ~/.xinitrc или ~/.xsession.)
4. Альтернативно, добавьте путь к шрифтам в файл /usr/local/etc/fonts/local.conf в формате XML:
   ```
   <dir>/usr/local/lib/X11/fonts/URW</dir>
   ```

• **Установка шрифтов TrueType**:

1. Убедитесь, что модуль freetype включён в конфигурации X-сервера. Добавьте в /etc/X11/xorg.conf в секцию Module:
   ```
   Load "freetype"
   ```

2. Создайте каталог для шрифтов TrueType, например:
   ```
   # mkdir /usr/local/lib/X11/fonts/TrueType
   ```
3. Скопируйте файлы шрифтов TrueType (в формате UNIX/MS-DOS/Windows) в этот каталог.
4. Установите утилиту ttmkfdir из порта x11-fonts/ttmkfdir:
   ```
   # cd /usr/ports/x11-fonts/ttmkfdir
   # make install clean
   ```
5. Создайте файл fonts.dir в каталоге со шрифтами:
   ```
   # cd /usr/local/lib/X11/fonts/TrueType
   # ttmkfdir -o fonts.dir
   ```
6. Добавьте каталог со шрифтами TrueType к маршруту поиска шрифтов:
   - Выполните команды:
     ```
     # xset fp+ /usr/local/lib/X11/fonts/TrueType
     # xset fp rehash
     ```
   - Или добавьте в /etc/X11/xorg.conf в секцию Files:
     ```
     FontPath "/usr/local/lib/X11/fonts/TrueType/"
     ```
   - Или добавьте в /usr/local/etc/fonts/local.conf:
     ```
     <dir>/usr/local/lib/X11/fonts/TrueType</dir>
     ```
7. Для включения антиалиасинга создайте или отредактируйте файл /usr/local/etc/fonts/local.conf в формате XML, добавив настройки, например:
   ```
   <?xml version="1.0"?>
   <!DOCTYPE fontconfig SYSTEM "fonts.dtd">
   <fontconfig>
       <dir>/usr/local/lib/X11/fonts/TrueType</dir>
       <match target="font">
           <test name="size" compare="less"><double>14</double></test>
           <edit name="antialias" mode="assign"><bool>false</bool></edit>
       </match>
   </fontconfig>
   ```
8. Перестройте кэш шрифтов:
   ```
   # fc-cache -f
   ```

• **Дополнительные настройки для антиалиасинга**:

1. Для моноширинных шрифтов, если возникают проблемы с межсимвольным интервалом (например, в KDE), добавьте в /usr/local/etc/fonts/local.conf:
   ```
   <match target="pattern" name="family">
       <test qual="any" name="family"><string>fixed</string></test>
       <edit name="family" mode="assign"><string>mono</string></edit>
   </match>
   <match target="pattern" name="family">
       <test qual="any" name="family"><string>console</string></test>
       <edit name="family" mode="assign"><string>mono</string></edit>
   </match>
   <match target="pattern" name="family">
       <test qual="any" name="family"><string>mono</string></test>
       <edit name="spacing" mode="assign"><int>100</int></edit>
   </match>
   ```
2. Для проблемных шрифтов, таких как Helvetica, добавьте:
   ```
   <match target="pattern" name="family">
       <test qual="any"
   name="family"><string>Helvetica</string></test>
       <edit name="family" mode="assign"><string>sans-
   serif</string></edit>
   </match>
   ```
3. Для LCD-дисплеев включите разбиение точек (subpixel rendering):
   ```
   <match target="font">
       <test qual="all" name="rgba"><const>unknown</const></test>
   ```

```
        <edit name="rgba" mode="assign"><const>rgb</const></edit>
    </match>
```

(Замените rgb на bgr, vrgb или vbgr в зависимости от типа дисплея.)

4. Перестройте кэш шрифтов после изменений:
   ```
   # fc-cache -f
   ```

46. Предложите алгоритм установки FreeBSD.

1. **Подготовка установочного носителя**:
   - Скачайте образ установочного носителя FreeBSD (CD, DVD или USB) с официального сайта FreeBSD (ftp://ftp.FreeBSD.org) или используйте предоставленный CD-диск.
   - Проверьте целостность загруженного образа, сравнив контрольную сумму SHA256 с файлом CHECKSUM.SHA256 с помощью команды:
     ```
     sha256 FreeBSD-<version>-RELEASE-<arch>-<type>.img
     ```
   - Для записи образа на USB-накопитель:
     - В FreeBSD используйте команду dd:
       ```
       dd if=FreeBSD-<version>-RELEASE-<arch>-memstick.img of=/dev/da0 bs=64k
       ```
     - В Windows используйте утилиту, например, Win32DiskImager, выбрав образ и USB-устройство, затем нажав "Write".
   - Для CD/DVD запишите образ с помощью программы записи дисков, например, cdrecord в FreeBSD:
     ```
     cdrecord dev=/dev/cd0 FreeBSD-<version>-RELEASE-<arch>-disc1.iso
     ```
2. **Настройка BIOS/UEFI**:
   - Перезагрузите компьютер и войдите в настройки BIOS/UEFI (обычно клавиши F2, Del, F10, F11, F12 или Esc).
   - Измените порядок загрузки, установив приоритет для CD/DVD или USB-накопителя.
   - Сохраните изменения и перезагрузите компьютер с вставленным установочным носителем.
3. **Запуск установки**:
   - При загрузке с носителя появится меню загрузчика FreeBSD. Нажмите Enter для продолжения или дождитесь автоматического запуска (10 секунд).
   - Для специфических платформ:
     - **Macintosh PowerPC**: Удерживайте клавишу C во время загрузки или используйте Open Firmware, введя:
       ```
       boot cd:,\ppc\loader cd:0
       ```
     - **Sparc64**: Войдите в PROM (OpenFirmware), нажав L1+A или Stop+A, затем введите:
       ```
       boot cdrom
       ```
4. **Выбор раскладки клавиатуры**:
   - В программе установки bsdinstall выберите раскладку клавиатуры, если предложено. Используйте клавиши навигации и Enter для выбора подходящей раскладки (например, "United States of America ISO-8859-1" для стандартной).
   - Нажмите Esc, чтобы использовать раскладку по умолчанию, если не уверены.
5. **Указание имени хоста**:
   - Введите полное имя хоста (fully-qualified), например, machine3.example.com.
6. **Выбор устанавливаемых компонентов**:
   - Выберите дополнительные компоненты для установки (ядро и базовая система устанавливаются всегда). Возможные компоненты:

- doc – дополнительная документация.
- games – традиционные BSD-игры.
- lib32 – библиотеки для 32-битных приложений на 64-битной системе.
- ports – коллекция портов для установки программ.
- src – исходный код системы (требует 1 ГБ + 5 ГБ для пересборки).

- Отметьте нужные компоненты с помощью клавиши пробела и подтвердите выбор.

7. **Разбиение жесткого диска**:
   - Выберите способ разбиения: шаблонное (guided), ручное (manual) или через командный интерпретатор (shell).
   - **Шаблонное разбиение**:
     - Выберите диск для установки (если их несколько).
     - Выберите, использовать весь диск (Entire Disk) или только часть (Partition).
     - Подтвердите созданные разделы, просмотрев результат. Используйте Revert для отмены или Auto для повторного автоматического разбиения.
   - **Ручное разбиение**:
     - Выберите диск (например, ada0) и нажмите Create.
     - Выберите схему разбиения (рекомендуется GPT для современных систем, MBR для совместимости со старыми ОС, например, Windows XP).
     - Создайте разделы, указав тип, размер, точку монтирования и метку. Минимально:
       - freebsd-boot (512 КБ, загрузочный код).
       - freebsd-ufs (2 ГБ, точка монтирования /, метка, например, exrootfs).
       - freebsd-swap (4 ГБ, метка, например, exswap).
     - При необходимости создайте дополнительные разделы (/var, /tmp, /usr) с соответствующими размерами и метками.
     - Подтвердите разбиение, выбрав Finish.
   - **Разбиение с использованием fdisk**:
     - Выберите диск (например, ad0).
     - Для использования всего диска нажмите A (Use Entire Disk), удаляя существующие слайсы.
     - Выберите созданный слайс и нажмите S, чтобы сделать его загрузочным.
     - Для освобождения места удалите существующий слайс (D), затем создайте новый (C), указав размер.
     - Сохраните изменения, нажав Q.
8. **Подтверждение изменений**:
   - Подтвердите запись изменений на диск, выбрав Commit. До этого момента установку можно прервать без изменений данных.
9. **Процесс установки**:
   - Установщик отформатирует разделы с помощью newfs.
   - Если используется носитель bootonly, настройте сетевое соединение (Ethernet, DHCP или статические настройки: IP-адрес, маска подсети, шлюз, DNS, доменное имя) и выберите зеркало сайта для загрузки файлов.
   - Установщик загрузит, проверит и распакует файлы дистрибутива.
10. **Послеустановочная настройка**:
    - Установите пароль для пользователя root. Введите пароль дважды (символы не отображаются).
    - Настройте дополнительные параметры (если требуется) через финальное конфигурационное меню.
11. **Перезагрузка**:
    - Перезагрузите компьютер, выбрав соответствующую опцию в установщике.
    - Извлеките установочный носитель, чтобы загрузиться с жесткого диска.

47. Предложите методы разделения дискового пространства.

1. **Шаблонное (Guided) разделение**:
   - **Описание**: Этот метод автоматически создаёт разделы на диске, минимизируя вмешательство пользователя. Подходит для начинающих или стандартных установок.
   - **Процесс**:
     - В программе установки bsdinstall выберите опцию guided.
     - Если в системе несколько дисков, выберите целевой.
     - Решите, использовать весь диск (Entire Disk) или только часть (Partition).
     - Установщик автоматически создаёт стандартные разделы, например:
       - freebsd-boot (для загрузочного кода).
       - freebsd-ufs (корневая файловая система /).
       - freebsd-swap (раздел подкачки).
     - Просмотрите результат и подтвердите выбор, нажав Finish.
   - **Преимущества**:
     - Простота и автоматизация.
     - Подходит для стандартных конфигураций.
   - **Недостатки**:
     - Ограниченная гибкость в настройке размеров и структуры разделов.
2. **Ручное (Manual) разделение**:
   - **Описание**: Позволяет пользователю вручную настроить разделы, выбирая схему разбиения и параметры каждого раздела. Подходит для опытных пользователей, которым нужна специфическая структура.
   - **Процесс**:
     - В bsdinstall выберите опцию manual.
     - Выберите диск (например, ada0) и нажмите Create.
     - Выберите схему разбиения:
       - GPT (GUID Partition Table) – современный стандарт, поддерживает до 128 разделов, рекомендуется для новых систем.
       - MBR (Master Boot Record) – традиционная схема, ограничена 4 первичными разделами, подходит для совместимости со старыми ОС (например, Windows XP).
       - APM (Apple Partition Map) – для PowerPC Macintosh.
       - BSD (BSD Labels) – без MBR или в режиме "dangerously dedicated".
       - PC98 – для NEC PC-98.
       - VTOC8 (Volume Table Of Contents) – для Sun SPARC64/UltraSPARC.
     - Создайте разделы, указав тип, размер, точку монтирования и метку:
       - Типы разделов: freebsd-boot, freebsd-ufs, freebsd-swap.
       - Размер: задаётся в КБ, МБ, ГБ (например, кратный 1 МБ для выравнивания).
       - Точка монтирования: например, /, /var, /tmp, /usr.
       - Метка: уникальное имя (например, exrootfs, exswap).
     - Пример традиционного разбиения:
       - freebsd-boot (512 КБ).
       - freebsd-ufs (2 ГБ, /, exrootfs).
       - freebsd-swap (4 ГБ, exswap).
       - freebsd-ufs (2 ГБ, /var, exvarfs).
       - freebsd-ufs (1 ГБ, /tmp, extmpfs).
       - freebsd-ufs (остаток, /usr, exusrfs).
     - Подтвердите разбиение, выбрав Finish, и сохраните изменения (Commit).
   - **Преимущества**:
     - Полный контроль над структурой разделов.

        - Возможность создания сложных конфигураций.
    - **Недостатки**:
        - Требует опыта и понимания структуры файловых систем.
3. **Разделение с использованием FDisk**:
    - **Описание**: Использует утилиту fdisk для управления разделами в стиле MBR. Подходит для более старых систем или случаев, когда требуется совместимость с другими ОС.
    - **Процесс**:
        - В bsdinstall выберите опцию manual или shell, чтобы запустить fdisk.
        - Выберите диск (например, ad0).
        - Просмотрите текущие слайсы (разделы).
        - Для использования всего диска:
            - Нажмите A (Use Entire Disk) – существующие слайсы удаляются, создаётся один большой слайс для FreeBSD.
            - Выберите созданный слайс и нажмите S, чтобы сделать его загрузочным.
        - Для частичного разбиения:
            - Удалите существующий слайс (D).
            - Создайте новый слайс (C), указав размер.
        - Сохраните изменения (Q).
    - **Преимущества**:
        - Простота для базового разбиения.
        - Совместимость с MBR.
    - **Недостатки**:
        - Ограничение MBR (4 первичных раздела).
        - Меньшая гибкость по сравнению с GPT.
4. **Разделение через командный интерпретатор (Shell)**:
    - **Описание**: Позволяет использовать низкоуровневые утилиты (gpart, fdisk, bsdlabel) для разбиения диска. Подходит для продвинутых пользователей, которым нужны специфические настройки.
    - **Процесс**:
        - В bsdinstall выберите опцию shell.
        - Используйте команды, например:
            - gpart для работы с GPT:
```
gpart create -s gpt ada0
gpart add -t freebsd-boot -s 512K ada0
gpart add -t freebsd-ufs -s 2G ada0
gpart add -t freebsd-swap -s 4G ada0
```
            - fdisk для работы с MBR (аналогично описанному выше).
            - bsdlabel для создания BSD-меток внутри слайсов.
        - После создания разделов выйдите из shell (exit) и продолжите установку.
    - **Преимущества**:
        - Максимальная гибкость.
        - Доступ к низкоуровневым инструментам.
    - **Недостатки**:
        - Требует глубоких знаний.
        - Высокий риск ошибок.
5. **Предварительное разделение с помощью сторонних утилит**:
    - **Описание**: Разделение диска до установки с использованием внешних инструментов, таких как GParted.
    - **Процесс**:
        - Используйте загрузочный диск с GParted Live или другой утилитой.

- Создайте или измените разделы:
  - Освободите место, уменьшив существующие разделы (например, Windows).
  - Создайте новые разделы для FreeBSD (например, 5 ГБ для базовой установки с графикой).
- Сохраните изменения и начните установку FreeBSD, выбрав подготовленные разделы.

- **Преимущества**:
  - Удобство для сложных мультизагрузочных конфигураций.
  - Визуальный интерфейс.
- **Недостатки**:
  - Требует отдельного инструмента.
  - Риск потери данных при ошибках.

48. Предложите варианты применения межсетевого экрана открытого типа.

Межсетевой экран открытого типа (устанавливается с параметром firewall_type="open" в /etc/rc.conf) пропускает весь входящий и исходящий трафик без фильтрации. Варианты применения включают:

1. **Тестовая среда**: Использование на системах, где требуется полная доступность сети для отладки или тестирования приложений без ограничений.
2. **Локальная сеть без угроз**: Применение в доверенной внутренней сети (например, в корпоративной LAN), где нет необходимости в фильтрации трафика.
3. **Обучающие системы**: Установка на учебных серверах или рабочих станциях, где студенты или администраторы изучают сетевые процессы без ограничений.
4. **Временная конфигурация**: Использование на начальном этапе настройки сети, пока не разработаны и не применены конкретные правила фильтрации.

49. Предложите варианты применения межсетевого экрана закрытого типа.

Межсетевой экран закрытого типа (например, с набором правил, где по умолчанию блокируется весь трафик, а разрешение задаётся явно, как в примерах /etc/ipfw.rules) обеспечивает высокую безопасность. Варианты применения включают:

1. **Защита веб-сервера**: Разрешение только трафика на порт 80 (HTTP) или 443 (HTTPS) с лимитом соединений (например, limit src-addr 2), как показано в правилах ipfw add 00400 allow tcp from any to me 80 in via rl0 setup limit src-addr 2, для защиты от перегрузки.
2. **Сетевой шлюз с NAT**: Использование с правилами трансляции адресов (например, ipfw add 500 divert natd ip from any to any out via rl0) для безопасного доступа внутренней сети с приватными IP к Интернету.
3. **Ограничение доступа к критическим сервисам**: Блокировка нежелательного трафика (например, ipfw add 00310 deny icmp from any to any in via rl0 для запрета пинга) и разрешение только необходимых портов (SSH на 22, SMTP на 25 и т.д.).
4. **Защита от внешних атак**: Применение для серверов, подключённых к Интернету, с правилами блокировки немаршрутизируемых сетей (ipfw add 00300 deny all from 192.168.0.0/16 to any in via rl0) и логированием нежелательного трафика (ipfw add 00999 deny log all from any to any) для анализа атак.

50. Приведите алгоритм настройки межсетевого экрана

Межсетевой экран (firewall) является важнейшим элементом информационной безопасности при организации сетевого взаимодействия. В операционной системе FreeBSD одним из штатных средств реализации фильтрации сетевого трафика является IPFW — модульный, расширяемый и настраиваемый межсетевой экран, реализованный на уровне ядра. Его возможности включают фильтрацию пакетов, поддержку состояния соединений, NAT, управление пропускной способностью и пр.

Настройка IPFW осуществляется поэтапно и требует как включения подсистемы, так и определения набора правил фильтрации.

1. Проверка наличия IPFW и его установка

В первую очередь необходимо убедиться, что система поддерживает IPFW. В большинстве версий FreeBSD IPFW присутствует как модуль ядра (ipfw.ko) и может быть загружен динамически:

```
kldstat | grep ipfw
```

Если модуль не загружен, его можно активировать:

```
kldload ipfw
```

Для обеспечения автоматической загрузки IPFW при старте системы следует добавить строку в файл /boot/loader.conf:

```
ipfw_load="YES"
```

2. Активация IPFW в системе

После загрузки модуля необходимо включить сам фаервол в системных конфигурациях. Для этого в файл /etc/rc.conf` добавляется следующая строка:

```
firewall_enable="YES"
```

По умолчанию, FreeBSD может использовать предопределённые наборы правил (open, client, simple, closed), однако для большей гибкости предпочтительно использовать индивидуально составленные правила:

```
firewall_type="open"
```

Если используется пользовательский скрипт правил, указывается путь к нему:

```
firewall_script="/usr/local/etc/ipfw.rules"
```

3. Создание набора правил фильтрации

Правила IPFW могут быть оформлены в виде shell-скрипта. Они описываются в строгом порядке и имеют приоритет по номеру (от меньших к большим). Пример базового набора правил:

```
#!/bin/sh
ipfw -q -f flush
#удаляет все ранее заданные правила для очистки текущей конфигурации.
ipfw add 100 allow all from any to any via lo0
#разрешает весь трафик через loopback-интерфейс
ipfw add 110 deny all from any to 127.0.0.0/8.
#предотвращает спуфинг (подмену IP-адресов) на адреса локального интерфейса.
ipfw add 200 allow tcp from any to me 22 in
#разрешает входящие SSH-подключения к локальному хосту (порт 22).
ipfw add 300 allow tcp from me to any out keep-state
ipfw add 310 allow udp from me to any out keep-state
ipfw add 320 allow icmp from me to any out keep-state
#разрешают исходящие TCP, UDP и ICMP соединения с сохранением состояния
ipfw add 400 check-state
#проверка состояний соединений.
ipfw add 500 deny all from any to any
```

```
#базовое правило "по умолчанию": блокирует весь остальной трафик,
не подпадающий #под разрешающие правила.
```

4. Применение правил и перезагрузка IPFW

После написания правил необходимо либо выполнить скрипт вручную, либо перезапустить службу:

```
service ipfw restart
```

Или загрузить правила вручную:

```
sh /usr/local/etc/ipfw.rules
```

5. Тестирование работоспособности правил

На этапе тестирования необходимо проверить, что:

- Локальные приложения продолжают функционировать корректно (например, DNS-запросы, SSH).
- Запрещённые соединения действительно блокируются.
- Доступ к внешним ресурсам (через ICMP или TCP) возможен при наличии соответствующих разрешений.

Например

```
ping 8.8.8.8          # проверка ICMP
telnet myhost 22      # проверка TCP (SSH)
```

Также можно использовать ipfw show для просмотра количества срабатываний каждого правила.

51. Предложите вариант получения доступа к удаленным почтовым ящикам по протоколам POP и IMAP.

Доступ к удалённым почтовым ящикам осуществляется посредством протоколов POP (Post Office Protocol) и IMAP (Internet Message Access Protocol), которые определяют методы взаимодействия почтового клиента с сервером электронной почты.

Протокол POP, в частности версия POP3, предназначен для односторонней передачи сообщений с сервера на клиентское устройство. В ходе сессии клиент устанавливает соединение с сервером, осуществляет аутентификацию посредством учётных данных (логин и пароль), после чего загружает электронные сообщения на локальный носитель. При этом возможна конфигурация, предусматривающая удаление сообщений с сервера после их загрузки, что обеспечивает освобождение серверного пространства и исключает дублирование писем. Данный протокол ограничен функционально и не предусматривает поддержку работы с папками или синхронизации состояния сообщений.

Протокол IMAP обеспечивает двустороннюю синхронизацию между клиентом и сервером, позволяя пользователю работать с почтой непосредственно на сервере. После установления соединения и успешной аутентификации клиент получает информацию о структуре почтового ящика, включая перечень папок и статусы сообщений (прочитано, удалено, помечено и т. п.). IMAP поддерживает операции с почтовыми папками, а также частичную загрузку сообщений, что повышает эффективность работы с большим объёмом данных и обеспечивает единство состояния почтового ящика при работе с различных устройств. Как правило, соединение осуществляется на стандартных портах 143 (нешифрованное соединение) либо 993 (с использованием SSL/TLS).

Для установления соединения с сервером по обоим протоколам требуется указать адрес сервера, порт (обычно 110 для POP3, 143 для IMAP), а также аутентификационные данные — логин и пароль пользователя.

52. Предложите вариант получения доступа к локальным почтовым серверам.

Доступ к локальной почте в операционных системах Unix/Linux обычно осуществляется через системные директории, такие как /var/mail/ или /var/spool/mail/. В этих каталогах хранятся почтовые ящики пользователей в виде файлов, куда почтовый агент доставляет входящие сообщения. Утилиты командной строки, такие как mail, mutt, pine и другие, читают почту непосредственно из этих файлов, обеспечивая взаимодействие пользователя с локальными сообщениями.

Существует несколько распространённых форматов хранения почты:

- mbox — традиционный формат, в котором все письма пользователя хранятся последовательно в одном файле (/var/mail/username).
- Maildir — современный формат, в котором каждое письмо сохраняется в отдельном файле внутри структуры папок cur/, new/ и tmp/ в домашнем каталоге пользователя (~/Maildir).
- MH — менее распространенный формат, предполагающий хранение каждого сообщения в отдельном файле в каталоге, соответствующем почтовому ящику.

Для корректного доступа и предотвращения повреждений почтового файла используются механизмы блокировки, такие как flock или dotlock, которые исключают одновременную запись в почтовый ящик несколькими процессами.

Почтовые клиенты могут использовать переменные окружения (например, `$MAIL`) или собственные конфигурационные файлы (~/.muttrc, ~/.pinerc) для указания пути к почтовому ящику, а также формата хранения сообщений.

В процессе доставки почты локальные агенты доставки (MDA), такие как procmail или maildrop, получают сообщения от почтовых серверов (MTA, например Postfix, Sendmail или Exim) и размещают их в соответствующих почтовых ящиках, реализуя при этом фильтрацию и сортировку писем.

Для поддержания работоспособности и контроля за размером локальных почтовых ящиков применяются механизмы ротации и квотирования. Ротация обеспечивает архивирование или удаление устаревших сообщений, предотвращая переполнение дискового пространства, а квоты ограничивают максимальный размер почтового ящика пользователя.

53. Предложите вариант установки sendmail как программы по умолчанию.

Почтовый сервис (демон) (Mail Transfer Agent, MTA) – это программное обеспечение, которое отвечает за передачу почтовых сообщений от одного почтового сервера к другому.

Sendmail — это один из самых старых и широко используемых агентов передачи почты (MTA, Mail Transfer Agent) в Unix-подобных операционных системах, таких как FreeBSD, Linux и других.

Установка и настройка sendmail:

1. Установка sendmail:

```
sudo pkg install sendmail
```

Если sendmail уже установлен, то обновите его:

```
sudo pkg upgrade sendmail
```

2. Настройка конфигурационного файлов:

• /etc/mail/access: В этом файле определяются правила доступа к sendmail для различных

IP-адресов или доменов.

• /etc/mail/aliases: Файл aliases используется для определения альтернативных имен для почтовых ящиков. Например, это может быть привязка к имени пользователя системы или группы пользователей, чтобы все письма, адресованные алиасу, были доставлены в реальные почтовые ящики.

• /etc/mail/local-host-names: Список хостов, для которых sendmail принимает почту. В этом файле перечисляются все локальные домены и хосты, для которых Sendmail является локальным MTA. Sendmail будет принимать и обрабатывать почту только для доменов, перечисленных в этом файле.

• /etc/mail/mailer.conf: Настройки почтовой программы. Файл используется для определения настроек почтовых программ (MTA) и методов доставки. Он задает параметры конфигурации, такие как используемые почтовые агенты и их приоритеты.

• /etc/mail/mailertable: Таблица доставки почтовой программы. Этот файл используется для задания специфических правил маршрутизации почты для адресов или доменов. Он позволяет перенаправлять почту для определенных адресов на другие почтовые серверы или задавать специфические параметры доставки.

• /etc/mail/sendmail.cf: Основной файл настройки sendmail. Файл определяет, как Sendmail взаимодействует с почтовыми клиентами и другими MTA, настраивает маршрутизацию почты, определяет правила доставки и т.д.

• /etc/mail/virtusertable: Таблицы виртуальных пользователей и доменов. Файл используется для создания виртуальных пользователей или доменов, которые не существуют локально на сервере. Это позволяет Sendmail перенаправлять почту для таких виртуальных адресов на реальные пользователи или домены.

3. Включение sendmail в /etc/rc.conf: `sendmail_enable="YES"`

Или с помощью команды:

```
sysrc sendmail_enable=YES
```

4. Запуск sendmail:

```
service sendmail start
```

5. Проверка работоспособности sendmail:

```
echo "Тестовое сообщение" | mail -s "Тестовое письмо" адрес_получателя @example.com
```

6. Проверка логов sendmail Для уточнения проблем:

```
tail -f /var/log/maillog
```

54. Предложите вариант настройки почты для локального домена.

1. Указать домен в sendmail.mc с помощью MASQUERADE_DOMAIN

Чтобы все исходящие письма имели одинаковый домен (например, example.com вместо имени локального хоста вроде localhost.localdomain), нужно использовать маскарадинг (masquerading):

Для этого необходимо

- Открыть файл конфигурации sendmail.mc (обычно он находится в /etc/mail/)
- Добавить или раскомментировать следующие строки:

```
FEATURE(`masquerade_envelope')dnl
# замещает домен в SMTP-оболочке (envelope).
FEATURE(`allmasquerade')dnl
# замещает также адреса в заголовках To: и Cc:
MASQUERADE_AS(`example.com')dnl
# домен, который будет подставляться во все исходящие сообщения
MASQUERADE_DOMAIN(`example.com')dnl
# список доменов, для которых применяется маскарадинг.
```

2. Настроить local-host-names с нужным именем хоста

Файл local-host-names определяет, какие домены считаются локальными и для каких доменов Sendmail будет принимать почту.

Для его настройки необходимо:

- Открыть файл /etc/mail/local-host-names
- Указать в нем доменное имя, которое необходимо обслуживать:

```
example.com
mail.example.com
```

3. Скомпилировать конфигурацию и перезапустите sendmail

После внесения изменений в sendmail.mc необходимо скомпилировать его в sendmail.cf, который используется непосредственно Sendmail. Для этого:

- Выполнить команду компиляции:

```
cd /etc/mail
sudo make
```

- Перезапустить службу Sendmail, чтобы применить изменения:

```
sudo service sendmail restart
```

- При необходимости перезапустить также sm-msp-queue (клиентскую очередь сообщений):

```
sudo service sendmail submit restart
```

После выполнения этих шагов все исходящие письма будут отображаться как отправленные с домена example.com, и Sendmail будет принимать сообщения, адресованные на example.com и другие указанные в local-host-names имена.

55. Предложите вариант настройки SMTP через UUCP.

Если вы хотите подключить к интернет компьютер с FreeBSD, работающий в локальной сети. Компьютер с FreeBSD будет почтовым шлюзом для локальной сети. Один способ, чтобы это реализовать – использование UUCP.

SMTP (Simple Mail Transfer Protocol) — это протокол передачи почты, который используется для отправки электронной почты между серверами.

UUCP (Unix-to-Unix Copy Protocol) — это протокол для передачи файлов и команд между компьютерами, работающими под операционными системами Unix или Unix-подобными системами.

Использование UUCP для SMTP (Simple Mail Transfer Protocol) возможно, если требуется настроить отправку и получение электронной почты между системами через модемные соединения или через другие точечные каналы, которые не имеют постоянного или надежного сетевого соединения.

Настройка Sendmail для использования UUCP:

1. Использование различных файлов конфигурации

Редактирование файла /etc/mail/sendmail.cf вручную сложно и не рекомендуется. Вместо этого sendmail версии 8 использует препроцессор m4 для генерации конфигурационных файлов. Примеры файлов настройки m4 расположены в каталоге /usr/share/sendmail/cf.

2. Генерация конфигурационных файлов через m4:

Для создания конфигурационного файла .mc можно использовать примеры из каталога /usr/share/sendmail/cf/cf, например, foo.mc. После редактирования этого файла его можно преобразовать в sendmail.cf следующим образом:

```
# cd /etc/mail
# make foo.cf
# cp foo.cf /etc/mail/sendmail.cf
```

3. Пример .mc файла для настройки UUCP:

```
VERSIONID(`Your version number')
OSTYPE(bsd9.1)
FEATURE(accept_unresolvable_domains)
FEATURE(nocanonify)
FEATURE(mailertable, `hash -o /etc/mail/mailertable')
define(`UUCP_RELAY',your.uucp.relay)
define(`UUCP_MAX_SIZE',200000)
define(`confDONT_PROBE_INTERFACES')
MAILER(local)
MAILER(smtp)
MAILER(uucp)
Cw your.alias.host.name
Cw youruucpnodename.UUCP
```

В типичном примере .mc используются следующие команды:

- `FEATURE(mailertable)`: Включает использование mailertable для маршрутизации почты.
- `define(UUCP_RELAY)`: Указывает основной сервер UUCP, через который будет отправляться почта.
- `MAILER(uucp)`: Определяет использование UUCP как одного из методов доставки.

4. Настройка mailertable

Файл mailertable (/etc/mail/mailertable) необходим для указания Sendmail, как обрабатывать почту для различных доменов или адресов через UUCP. Пример содержимого mailertable:

```
.example.com    uucp-dom:your.uucp.relay
```

После добавления записей в mailertable, необходимо создать базу данных для этой таблицы:

```
sudo makemap hash /etc/mail/mailertable < /etc/mail/mailertable
```

5. Применения изменений и тестирование:

После настройки и перезапуска Sendmail, следует протестировать отправку почты через UUCP, убедившись, что все настройки работают корректно.

56. Предложите варианты пользовательских почтовых программ для использования

Пользовательские почтовые программы (Mail User Agents, MUA) обеспечивают интерфейс между пользователем и системой доставки электронной почты. Они позволяют получать, просматривать, составлять и отправлять электронные письма. Ниже приведены популярные почтовые клиенты, доступные в текстовом и графическом вариантах.

1. Mail

mail — это базовая консольная утилита, входящая в состав большинства Unix-подобных операционных систем.

Интерфейс: текстовый (CLI).

Основные возможности: чтение и отправка писем, работа с локальной почтой (обычно /var/mail/username), поддержка вложений ограничена.

К преимуществам можно отнесть: лёгкость, всегда установлена, подходит для сценариев автоматической отправки писем (например, из скриптов).

Недостатки: минималистичный функционал, неудобна для работы с большим количеством сообщений.

2. Mutt

mutt — продвинутый текстовый почтовый клиент для терминала.

Интерфейс: текстовый, ncurses-интерфейс. Поддерживает POP3, IMAP, MIME,

PGP/GPG, работу с несколькими аккаунтами. Имеет возможности фильтрации, макросов и расширенной конфигурации. Может интегрироваться с внешними редакторами (например, Vim). Используется опытными пользователями и системными администраторами за счёт гибкости.

3. Pine (Program for Internet News and Email)
pine — текстовая почтовая программа, разработанная в Университете Вашингтона.

Интерфейс: текстовый. Предоставляет простой доступ к почте через IMAP и SMTP.

Проект устарел: в 2005 году разработка была прекращена.

4. Alpine
alpine — продолжение pine, разрабатываемое как свободное программное обеспечение.

Интерфейс: текстовый, удобный для новых пользователей. Поддерживает IMAP, POP3, SMTP, фильтрацию сообщений, PGP/GPG, вложения. Отличается стабильностью и хорошей поддержкой. Подходит для системных администраторов и пользователей, предпочитающих работу в терминале.

5. Thunderbird

Thunderbird — полнофункциональный графический почтовый клиент с открытым исходным кодом, разработанный Mozilla Foundation. Интерфейс: графический (GUI). Поддерживает IMAP, POP3, SMTP, адресную книгу, расширения, фильтры сообщений, шифрование. Имеет мощные инструменты для управления почтой, поиск, вкладки, спам-фильтры. Подходит для обычных пользователей, работающих в графической среде.

57. Предложите вариант отключения sendmail.

Почтовый сервис (демон) (Mail Transfer Agent, MTA) – это программное обеспечение, которое отвечает за передачу почтовых сообщений от одного почтового сервера к другому.

Sendmail — это один из самых старых и широко используемых агентов передачи почты (MTA, Mail Transfer Agent) в Unix-подобных операционных системах, таких как FreeBSD, Linux и других.

Для его полного отключения нужно прописать следующее в /etc/rc.conf:

```
sendmail_enable="NO"
sendmail_submit_enable="NO"
sendmail_outbound_enable="NO"
sendmail_msp_queue_enable="NO"
```

Если нужно отключить только сервис входящей почты, то достаточно этого:

```
sendmail_enable="NO"
```