1. Понятие операционной системы. Назначение операционных систем. Ресурсы, которыми управляет ОС.

ОС – комплекс системных программных средств, осуществляющий управление функционированием аппаратной вычислительной платформы с целью организации исполнения прикладных программ

Операционная система (ОС) — это комплекс программ, обеспечивающий интерфейс между аппаратными и программными ресурсами компьютера и пользователем или приложениями. В контексте Linux, ОС предоставляет основу для запуска широкого спектра программного обеспечения, обеспечивает многозадачность, защиту памяти, управление памятью и процессами, а также функции ввода-вывода.

Назначение ОС:

- Абстракция и управление аппаратурой: ОС скрывает сложность аппаратуры и предоставляет удобный интерфейс для использования ресурсов.

- Многозадачность: Позволяет одновременно выполнять несколько приложений, эффективно распределяя ресурсы процессора и памяти.

- Безопасность: Контролирует доступ к данным и ресурсам, предотвращая несанкционированное использование.

- Сетевые возможности: Обеспечивает сетевое взаимодействие, поддержку различных протоколов и интерфейсов.

- Управление файлами: Организация хранения и доступа к данным через файловую систему.

Ресурсы, которыми управляет ОС:

- Процессор (CPU): Распределение времени процессора между задачами и приложениями.

- Память (RAM и дисковая память): Управление и распределение памяти, обеспечение изоляции и защиты данных различных процессов.

- Устройства ввода-вывода: Интеграция и управление устройствами, такими как диски, принтеры, сетевые интерфейсы.

- Файловая система: Организация и управление доступом к данным на дисках.

- Пользователи и процессы: Управление пользователями, правами доступа и жизненным циклом процессов.

В контексте Linux, ОС использует ядро Linux, которое обеспечивает все вышеупомянутые функции и характеризуется своей модульностью, монолитностью (с возможностью загрузки драйверов в виде модулей), и широкими возможностями настройки. Linux также поддерживает различные файловые системы, сетевые протоколы и стандарт POSIX, что делает его мощным инструментом для разработчиков и системных администраторов.

2. Разновидности современных ОС. Их сходства и различия.

Современные операционные системы можно разделить на несколько категорий, в зависимости от их происхождения и основных характеристик:

I. UNIX-like (похожие на UNIX)

- Linux-like (базирующиеся на ядре Linux):

- Эти системы включают различные дистрибутивы Linux, такие как Debian (и его производные, включая Ubuntu, Linux Mint, Raspberry OS), RedHat (и его производные, такие как RHEL, OEL, Fedora, CentOS, Rocky Linux), Slackware (и связанные с ним SUSE, SLES, openSUSE), Gentoo (на котором основан ChromeOS), ArchLinux (и его производные, такие как Manjaro) и Android.

- Они сходятся в использовании ядра Linux и принципов открытого исходного кода, но различаются в системах управления пакетами, конфигурации системы и предназначении.

- BSD-like:

- Включают операционные системы, основанные на Berkeley Software Distribution, такие как FreeBSD (и другие BSD, включая NetBSD и OpenBSD), а также DarwinOS, который лежит в основе OSX/MacOS и iOS от Apple.

- Эти ОС делят общую философию UNIX, но имеют отличия в ядре и системных утилитах.

- SystemV:

- Включает традиционные коммерческие UNIX системы, такие как HP-UX, AIX от IBM, Solaris от Sun Microsystems, UnixWare и SCO.

- Они имеют собственные уникальные характеристики и управляющие команды, которые отличаются от Linux и BSD.

II. Microsoft OS

- Based on MS-DOS:

- Включает ранние версии Windows, такие как MS-DOS, Win3.x, Win95, Win98 и WinME, которые сейчас устарели.

- Windows NT:

- Современные версии Windows, начиная с Windows NT и до последних версий, таких как WinXP, Win7, Vista, Win10 и Win11.

- Эта линейка ОС основывается на ядре NT и предоставляет широкий спектр возможностей для домашних пользователей и корпоративных сред.

Сходства и различия между этими ОС заключаются в их дизайне и назначении. UNIX-like системы обычно считаются более модульными и гибкими с точки зрения настройки, а также поддерживают широкий спектр аппаратного обеспечения и программного обеспечения. Они часто используются в серверных и настольных средах, а также в области встроенных систем. Microsoft OS, с другой стороны, являются коммерческими продуктами, предлагающими интегрированный пользовательский опыт и широко используемыми в домашних и корпоративных средах.

3. Понятие прикладных программ (Applications): их структурные компоненты и функционирование.

Прикладные программы — это программы, предназначенные для выполнения конкретных пользовательских задач, таких как текстовый редактор, браузер, игры, специализированное программное обеспечение (например, для моделирования, бухгалтерского учета и т.д.).

Структурные компоненты:

- Пользовательский интерфейс (UI): Может быть графическим (GUI), текстовым или сенсорным, обеспечивает взаимодействие пользователя с программой.

- Логика приложения: Сердце приложения, содержит алгоритмы и процессы, необходимые для выполнения его функций.

- Данные и управление данными: Включают базы данных, файлы, API для взаимодействия с другими приложениями и системными ресурсами.

Функционирование:

1. Запуск: Инициирован пользователем или системой, приложение загружается в память.

2. Взаимодействие: Пользователь взаимодействует с приложением через его UI, вводя данные и команды.

3. Обработка: Приложение обрабатывает входные данные в соответствии с его логикой, возможно, взаимодействуя с другими системами или ресурсами.

4. Вывод: Приложение предоставляет результаты своей работы пользователю или другим системам, либо изменяет данные.

В контексте Linux, прикладные программы часто используют стандартные библиотеки и инструменты, доступные в системе, такие как GTK для графических приложений, различные языки программирования и их среды выполнения, и мощные инструменты

командной строки. Linux также поддерживает широкий спектр программного обеспечения, разработанного как сообществом, так и коммерческими поставщиками.

4. Вычислительные системы: понятие, назначение, разновидности, аппаратные структурные компоненты и их взаимодействие.

Вычислительные системы — это комплексы аппаратных и программных средств, предназначенные для автоматизации обработки информации, выполнения вычислений и управления данными.

Назначение:

- Автоматизация процессов обработки данных.

- Выполнение специфических вычислений и задач.

- Хранение, обработка и передача информации.

Разновидности:

- Персональные компьютеры

- Серверы

- Суперкомпьютеры

- Мобильные устройства

- Встраиваемые системы

Аппаратные структурные компоненты

- Центральный процессор (CPU): Выполняет инструкции программ и обрабатывает данные.

- Память: Хранит данные и инструкции для быстрого доступа.

- Устройства ввода-вывода: Позволяют взаимодействовать с внешним миром (клавиатура, мышь, экран, сетевые адаптеры).

- Хранилище данных: Устройства для долговременного хранения данных (жесткие диски, SSD, оптические носители).

- Шины и коммуникационные интерфейсы: Обеспечивают передачу данных между компонентами системы.

Их взаимодействие:

- CPU исполняет программы, обращаясь к памяти за данными и инструкциями.

- Память хранит временную информацию, необходимую процессору.

- Устройства ввода-вывода обеспечивают взаимодействие с пользователем и другими системами.

- Хранилище сохраняет данные и программы для долгосрочного использования.

- Шины и интерфейсы обеспечивают передачу данных между всеми компонентами.

5. Шинная организация вычислительных систем: шина памяти, шина ввода-вывода и их использование в процессе функционирования.

Шинная организация — это метод организации связи между различными компонентами вычислительной системы посредством одного или нескольких коммуникационных путей, называемых шинами.

Шина памяти обеспечивает передачу данных между ЦПУ и оперативной памятью. Она используется для загрузки данных в ЦПУ для обработки и отправки обратно в память обработанных данных или результатов.

Шина ввода-вывода соединяет ЦПУ и устройства ввода-вывода (например, жесткие диски, сетевые карты, периферийные устройства). Эта шина передает команды от ЦПУ к устройствам и данные между устройствами и оперативной памятью.

Использование в процессе функционирования:

- Шина памяти: Когда процессору нужно выполнить операцию, он загружает необходимые данные из оперативной памяти через шину памяти, обрабатывает их и затем отправляет результат обратно в память.

- Шина ввода-вывода: При работе с внешними устройствами, например, когда нужно считать данные с жесткого диска или отправить данные на принтер, используется шина ввода-вывода для передачи данных и команд.

Шинная организация позволяет упростить и стандартизировать обмен данными между компонентами, что облегчает разработку и поддержку вычислительных систем.

6. Память в составе вычислительных систем: основная память, регистры CPU, регистры контроллеров, адресация, адресное пространство.

Память в Linux организована сложно и многоуровнево:

- Основная память (RAM): Linux управляет RAM через механизмы виртуальной памяти, позволяя изолировать процессы друг от друга и оптимально распределять физическую память.

- Регистры CPU: Используются для непосредственного хранения данных и управления операциями процессора. Linux оптимизирует использование регистров для эффективной обработки задач.

- Регистры контроллеров: Интерфейс управления устройствами I/O, например, для дисков или сетевых карт. Драйверы в Linux обеспечивают взаимодействие с этими регистрами для выполнения операций ввода-вывода.

- Адресация: Включает в себя механизмы для определения расположения данных в памяти. В Linux широко используется виртуальная адресация, позволяющая программам работать с адресами, не заботясь о реальном расположении данных.

- Адресное пространство: Определяется как диапазон адресов, который может использовать процесс. Linux поддерживает большое адресное пространство, часто разделяемое на пользовательское пространство и пространство ядра, что обеспечивает изоляцию и безопасность.

7. Аппаратные и программные функции вычислительных систем. Понятие и назначение встроенного ПО.

Аппаратные и программные функции:

- Аппаратные функции: Включают всю физическую инфраструктуру компьютера: CPU, память, устройства ввода-вывода и др. Linux поддерживает широкий спектр аппаратных архитектур и предоставляет универсальные интерфейсы для их использования.

- Программные функции: Состоят из всех уровней программного обеспечения, от драйверов устройств до пользовательских приложений. В Linux это включает ядро, системные библиотеки, службы и приложения.

Встроенное ПО (Firmware):

- Является программным обеспечением, которое предоставляет низкоуровневый контроль над аппаратурой.

- В Linux, встроенное ПО часто используется для инициализации оборудования до загрузки операционной системы, например, в BIOS или UEFI.

- Во встроенных системах Linux, встроенное ПО может также включать загрузчики, такие как U-Boot, для инициализации системы.

8. Особенности разработки приложений для среды аппаратных платформ (Embedded programming).

Разработка для встроенных систем Linux имеет свои особенности:

- Ограниченные ресурсы: Встроенные системы часто имеют ограничения по памяти, мощности процессора и хранилищу. Это требует оптимизации производительности и эффективности.

- Реальное время: Многие встроенные приложения требуют гарантий времени отклика, что может потребовать использования реального времени (Real-Time) версий Linux.

- Специализированные устройства: Приложения могут взаимодействовать с нестандартным оборудованием, требующим специальных драйверов и программного обеспечения.

- Стабильность и надежность: Высокие требования к безотказной работе ввиду специфики использования встроенных систем.

9. Принцип разделения системного и прикладного ПО. Его значимость и преимущества.

Разделение системного и прикладного ПО важно для безопасности, удобства управления и стабильности системы:

- Безопасность: Изолирует системные процессы от прикладных, снижая риск повреждения системы из-за ошибок в приложениях или злонамеренных действий.

- Модульность: Упрощает обновление, модификацию и расширение как системного, так и прикладного ПО, не затрагивая другую часть системы.

- Стабильность: Системное ПО, управляющее ресурсами и процессами, может быть оптимизировано для высокой надежности и производительности.

В Linux это разделение проявляется в четком разграничении между ядром (системным ПО) и пользовательским пространством, где выполняются все прикладные программы.

10. Функционирование приложений в среде ОС: ядро, процессы и их взаимодействие, приложения и службы.

Ядро Linux управляет аппаратными ресурсами, процессами, памятью, файловой системой и сетевыми операциями. Оно является мостом между физической аппаратурой и прикладным программным обеспечением.

- Процессы: Каждое приложение или служба в Linux запускается в виде процесса с уникальным идентификатором (PID). Процессы изолированы друг от друга, имеют свое виртуальное адресное пространство и ресурсы.

- Взаимодействие: Процессы могут взаимодействовать через системные вызовы, сигналы, IPC (межпроцессное взаимодействие), сокеты и файлы.

- Приложения и службы: Приложения обеспечивают функциональность для пользователя, в то время как службы работают в фоновом режиме, поддерживая основные функции системы, такие как логирование, планирование задач и сетевые соединения.

Linux предоставляет мощные инструменты для управления процессами, включая приоритеты, управление ресурсами и изоляцию, что делает его мощной и гибкой операционной системой как для пользовательских приложений, так и для критически важных системных служб.

11. Структура ОС: системный том и его содержимое, процесс установки, дистрибутив, пакеты, репозитории пакетов, обновление ОС.

В Linux, системный том обычно включает в себя следующие основные компоненты:

- /boot: Содержит ядро Linux и все необходимое для его загрузки, включая загрузчик GRUB.

- /bin и /sbin: Основные исполняемые файлы системы и системные утилиты.

- /etc: Конфигурационные файлы системы.

- /lib: Основные библиотеки системы и модули ядра.

- /usr: Дополнительные приложения и файлы.

- /var: Переменные файлы, такие как логи и кэш.

- /home: Домашние каталоги пользователей.

Процесс установки в Linux включает выбор дистрибутива, разбиение диска, установку ядра и системных файлов, настройку загрузчика и первоначальную конфигурацию.

Дистрибутивы Linux, такие как Ubuntu, Fedora, или Debian, предоставляют предварительно собранные и настроенные пакеты программного обеспечения, инструменты управления пакетами и специфические для дистрибутива настройки.

Пакеты и репозитории: Большинство дистрибутивов Linux используют системы управления пакетами (например, APT, YUM, или Pacman), которые позволяют устанавливать, обновлять и удалять программное обеспечение из репозиториев - сетевых хранилищ предварительно собранных и тестированных пакетов.

Обновление ОС: Системы управления пакетами также обеспечивают обновление всей системы и отдельных программ до последних версий, поддерживая систему безопасной и актуальной.

12. Структура ОС: ядро, драйверы, процессы, загрузка приложения/службы.

Структура Linux обычно включает:

- Ядро (Kernel): Центральная часть системы, управляющая аппаратными ресурсами, планированием процессов, системным вводом-выводом и сетевыми функциями.

- Драйверы: Модули ядра или встроенные в него компоненты, обеспечивающие поддержку различных типов аппаратного обеспечения.

- Процессы: Выполнение приложений и служб в изолированных контекстах. Linux поддерживает многозадачность и защиту памяти процессов.

- Загрузка приложения/службы: Приложения и службы загружаются в память и выполняются в виде процессов. Для этого используются системные вызовы и механизмы контроля доступа.

13. Состав программного обеспечения включаемого в состав ОС.

Операционная система Linux включает в себя следующие компоненты программного обеспечения:

- Ядро ОС (и набор модулей к ядру): Является основной частью ОС, которая постоянно находится в памяти и управляет ресурсами вычислительной системы.

- Программы начальной загрузки и инициализации ОС: Эти программы обеспечивают загрузку ядра, запуск процессов и служб, формирующих рабочую среду вычислительной системы.

- Командный интерпретатор (shell): Предоставляет пользователю интерфейс для управления запуском и выполнением других программ.

- Набор программ (команд) управления различными системными средствами ([admin-cmd]): Включает команды для управления внешними устройствами, дисковой памятью, процессами, пользователями и полномочиями, файлами, сетевыми средствами, службами.

- Набор программ (команд) и прикладных программ пользовательского уровня ([user-cmd]): Включает стандартные утилиты и приложения, такие как программы для копирования файлов или клиенты электронной почты.

- Система управления пакетами программ ([package-manager]): Управляет установкой, обновлением и удалением программных пакетов в ОС.

- Графическая оконная подсистема ([Windowing-system]): Управляет отображением графического интерфейса и взаимодействием с пользователем через окна.

- Графическая оболочка рабочего стола пользователя ([Desktop-environment]): Предоставляет визуальный интерфейс для управления системой и запуска приложений.

- Стандартные графические приложения ([GUI-tools]): Включает основные графические инструменты, такие как файловый менеджер, текстовый редактор и т.д.

- Подсистема разработки программного обеспечения ([Dev-tools]): Включает инструменты, необходимые для разработки программ, например компиляторы и среды разработки.

- Приложения ([Applications]): Включает разнообразные программы, такие как веб-браузеры и интегрированные среды разработки (IDE).

Эти компоненты совместно образуют функциональную и гибкую операционную систему, позволяющую пользователям и администраторам эффективно управлять ресурсами и выполнять задачи различной сложности.

14. Программный код и его варианты: native, эмулированный, интерпретируемый. Понятие компилятора и интерпретатора.

В Linux программный код может быть представлен в нескольких формах:

- Native (машинный код): Программы, скомпилированные непосредственно для архитектуры процессора системы. Исполняются с максимальной производительностью.

- Эмулированный: Код, который исполняется не на целевой архитектуре, а через прослойку эмуляции или виртуализации.

- Интерпретируемый: Код, который выполняется построчно с помощью интерпретатора (например, скрипты Bash или Python).

Компилятор преобразует исходный код программы в машинный код, оптимизированный для конкретной архитектуры и операционной системы. Интерпретатор выполняет исходный код программы непосредственно, переводя его инструкции на лету в машинные команды.

15. Структура машинного кода: инструкции, операнды, варианты адресации.

Машинный код состоит из:

- Инструкций: Команды, которые процессор исполняет для выполнения операций.

- Операндов: Данные, с которыми работают инструкции. Могут быть непосредственными значениями, адресами памяти или регистрами.

- Способы адресации: Методы, которые используются для определения местоположения операндов. Включают непосредственную, регистровую, индексную и другие формы адресации.

16. Варианты представления программного кода: абсолютный модуль, объектный модуль, исполняемый модуль, библиотека, разделяемая библиотека. Их назначение и использование.

В Linux программное обеспечение может быть представлено в виде:

- Абсолютный модуль: Программный код, скомпилированный в машинный код, готовый к выполнению на конкретном адресе в памяти.

- Объектный модуль: Промежуточное представление кода, содержащее машинные инструкции и метаданные, но еще не связанное с адресами памяти.

- Исполняемый модуль: Финальный продукт компиляции и линковки, содержащий машинный код, готовый к загрузке и выполнению в памяти.

- Библиотека: Набор кода и данных, которые могут быть использованы многими программами. В Linux используются как статические, так и динамические (разделяемые) библиотеки.

- Разделяемая библиотека: Библиотека, которая загружается в память один раз и может использоваться многими процессами, экономя ресурсы и память.

17. Конвейер создания исполняемых модулей: компилятор, архиватор, линкер. Варианты их применения в программных проектах. Роль объектных файлов.

Процесс создания исполняемых файлов в Linux обычно включает:

- Компилятор: Преобразует исходный код в объектные файлы, содержащие машинный код и информацию для связывания.

- Архиватор: Может использоваться для объединения нескольких объектных файлов в один архив (например, статическую библиотеку).

- Линкер: Связывает объектные файлы и библиотеки вместе, разрешая ссылки и адреса, создавая исполняемый файл или разделяемую библиотеку.

Объектные файлы играют ключевую роль, содержа машинный код и данные, а также метаинформацию для связывания и отладки.

18. Структура объектных файлов: секции памяти, символы, relocations, external reference

В Linux объектные файлы обычно используют формат ELF, и каждый такой файл содержит:

- Секции памяти (.text, .data, .bss, .rodata и др.):

  - .text: Содержит машинный код программы.

  - .data: Инициализированные данные, включая глобальные переменные.

  - .bss: Неинициализированные данные, которые должны быть инициализированы нулями.

- .rodata: Read-only данные, такие как строковые литералы.

- Символы: Имена функций и переменных, которые используются для организации и доступа к коду и данным. Символы могут быть локальными (видны только внутри файла) и глобальными (видны из других объектных файлов).

- Relocations (перемещения): Содержат информацию о том, как адреса в коде должны быть изменены, чтобы корректно ссылаться на функции и данные после загрузки программы в память.

- External references (внешние ссылки): Ссылки на символы, определенные в других модулях или библиотеках, которые должны быть разрешены во время линковки или загрузки.

19. Роль и функционирование компоновщика исполняемых файлов (linker). Процесс разрешения внешних ссылок.

Компоновщик (linker) выполняет критически важные задачи:

- Разрешение символов: Определяет адреса всех глобальных символов и обеспечивает их соответствие между различными объектными файлами.

- Обработка перемещений: Изменяет код и данные в объектных файлах в соответствии с их окончательными адресами в памяти.

- Создание исполняемого файла: Объединяет все необходимые объектные файлы и библиотеки в один исполняемый файл, размещая код и данные в соответствующие секции и исправляя ссылки.

Компоновщик также разрешает внешние ссылки, соединяя различные части программы и библиотеки в единое целое, и определяет, какие библиотеки должны быть загружены во время выполнения для динамических ссылок.

20. Разделяемые библиотеки, статическая и динамическая линковка. Их преимущества и недостатки.

Разделяемые библиотеки и механизмы линковки в Linux включают:

- Статическая линковка:

  - Преимущества: Независимость от библиотек системы, все необходимое содержится в самом исполняемом файле.

  - Недостатки: Большой размер исполняемого файла, необходимость перекомпиляции для обновления библиотек.

- Динамическая линковка:

  - Преимущества: Экономия дискового пространства и памяти, поскольку одна копия библиотеки может использоваться многими программами. Легкость обновления библиотек.

  - Недостатки: Зависимость от версий библиотек в системе, потенциальные проблемы с совместимостью и сложности с развертыванием.

21. Виды памяти в программном коде: статическая, динамическая (heap), программный стек. Способ реализации стека. Структура памяти процесса.

Память в программном коде в Linux разделяется на:

- Статическая память: Используется для хранения статических и глобальных переменных. Выделена на всё время работы программы.

- Динамическая память (куча): Позволяет программе выделять и освобождать память во время выполнения. Управляется системными вызовами, такими как `malloc` и `free` в C.

- Программный стек: Используется для хранения локальных переменных функций, информации о вызовах функций и возвращаемых адресах. Растет и уменьшается с вызовом и возвратом функций.

Стек обычно реализуется так, что его вершина уменьшается при добавлении элементов (так называемый "стек с убывающими адресами") и увеличивается при удалении элементов.

Структура памяти процесса включает текстовый сегмент (исполняемый код), сегмент данных (глобальные и статические переменные), кучу для динамически выделенной памяти, и стек.

22. Организация памяти в программе: статическая vs динамическая, программные секции .text, .data, .rodata, .bss.

Организация памяти в программе Linux включает:

- *Статическая память:*

  - .data: Секция для инициализированных глобальных и статических переменных.

  - .bss: Секция для неинициализированных глобальных и статических переменных, автоматически инициализируемых нулями.

  - .rodata: Секция для константных данных, таких как строковые литералы.

- *Динамическая память*: Управляется кучей и используется для динамического выделения и освобождения памяти во время выполнения программы.

- .text: Секция для машинного кода программы.

23. Поток исполнения программного кода. Главный цикл CPU. Управление последовательностью исполнения инструкций.

Поток исполнения в Linux контролируется операционной системой и ядром, которые распределяют время процессора между различными задачами:

- Главный цикл CPU (Fetch-Decode-Execute): Процессор поочередно извлекает инструкции из памяти (fetch), расшифровывает их (decode) и выполняет (execute).

- Управление последовательностью: Осуществляется с помощью счетчика команд (Program Counter, PC), который указывает на следующую исполняемую инструкцию. Контроль за

последовательностью может изменяться в результате условных и безусловных переходов, вызовов и возвратов из функций.

Эти процессы обеспечивают выполнение инструкций в правильном порядке и позволяют операционной системе эффективно управлять выполнением множества задач, обеспечивая многозадачность и отзывчивость системы.

24. Механизм прерываний. Его назначение и роль в ВС. Условия прерывания. Процедуры прерывания и возврата из прерывания. Векторы прерываний. ISR

Прерывание – процедура, исполняемая CPU core(почти самостоятельно) в качестве возникновения некоторого условия прерывания

Исполнение текущего потока приостанавливается и управление передается предопределенной процедуре обработки прерывания

Состояние текущего потока сохраняется таким образом, что обработчик сможет продолжить выполнение прерванного потока(если нужно)

Условиями прерывания:

- interrupts: внешние прерывания

- exceptions: события, возникающие в процессе исполнения программы

    * ошибки, детектируемые CPU

    * прерывания, вызываемые специальными инструкциями

Векторы прерываний и ISR (Interrupt Service Routine):

Векторы прерываний: Таблица, которая содержит адреса ISR для каждого типа прерывания. В Linux, эта таблица обычно находится в специально отведенной области памяти.

ISR: Специализированные функции в ядре ОС, обрабатывающие конкретные прерывания. В Linux, разработчики ядра и драйверов часто пишут свои ISR для обработки аппаратных прерываний от устройств.

25. Многозадачность в ОС, её назначение. Понятия контекст и переключение контекста. Потоки (threads). Диспетчеризация потоков исполнения.

Многозадачность — функциональность, позволяющая ОС выполнять параллельно несколько разных задач, что достигается за счет множества потоков исполнения (threads).

Контекст исполнения (execution context) изменяется в процессе работы задачи и включает в себя всю информацию, необходимую для выполнения потока, такую как значения регистров, счетчика команд и указателя стека.

Потоки (threads) в ОС представляют собой отдельные исполняемые единицы, которые могут выполняться независимо и параллельно (на системах с несколькими ядрами) или квазипараллельно (последовательно, но быстро переключаясь на одном ядре CPU).

Диспетчеризация потоков исполнения — это процесс управления порядком выполнения потоков ОС. Она осуществляется планировщиком ОС, который решает, какой поток будет выполнен следующим, основываясь на приоритете, времени CPU и других факторах.

Диспетчеризация включает сохранение контекста текущего потока исполнения и восстановление контекста следующего потока. Это позволяет ОС обеспечить иллюзию одновременности исполнения для пользователей и приложений, оптимизируя использование CPU и других ресурсов.

Таким образом, многозадачность играет важную роль в обеспечении эффективного и удобного использования вычислительных систем, позволяя одновременно выполнять множество различных процессов и потоков, обрабатывать внешние события и отклики, и поддерживать работу системных и пользовательских приложений без взаимного влияния и конфликтов.

26. Планировщик ОС: его роль и принципы функционирования. Очередь задач. Приоритеты.

1. Роль планировщика:

- Осуществляет распределение процессорного времени между задачами (tasks), определяя, какая задача будет использовать CPU.

2. Очередь задач:

- Задачи организованы в очереди в соответствии с их приоритетами и политиками планирования. Задачи с более высоким приоритетом находятся в верхней части очереди и выполняются в первую очередь. Каждая политика планирования может иметь свою собственную очередь.

3. Параметры, управляющие поведением scheduler:

- `scheduling policy`: Определяет, как задачи планируются для выполнения, и как они распределены в очередях:
  - `SCHED_DEADLINE`: Задачи управляются на основе их сроков выполнения.
  - `SCHED_FIFO`: Процессы обрабатываются в порядке очереди, без переключения до завершения или блокировки.
  - `SCHED_RR`: Схож с `SCHED_FIFO`, но добавляет кванты времени для циклического переключения между задачами с одинаковым приоритетом.
  - `SCHED_OTHER` / `SCHED_BATCH` / `SCHED_IDLE`: Используются для задач общего назначения с более сложным управлением очередями.

- `priority`: Определяет статический приоритет задачи в очереди.

- `nice`: Влияет на динамический приоритет задачи и её позицию в очереди.

4. Выбор задачи планировщиком:

- Планировщик выбирает задачи для выполнения, начиная с задачи с наивысшим приоритетом, а затем переходит к задачам с более низким приоритетом, при этом учитывая политику планирования и текущее состояние очередей.

Очередь задач и управление ею являются центральными элементами работы планировщика. Важно отметить, что планировщик постоянно переоценивает очередь, чтобы адаптироваться к изменяющимся условиям системы и потребностям процессов. Это гарантирует, что системные ресурсы распределены эффективно, а время отклика остается минимальным для всех типов задач.

27. Функционирование простого приоритетного планировщика.

Простой приоритетный планировщик в операционной системе Linux основан на присвоении каждому процессу приоритета, который используется для определения порядка их выполнения. Вот как он функционирует:

1. Присвоение приоритета:

Каждому процессу назначается статический приоритет, который может быть изменен в зависимости от типа задачи. Например, системные процессы или процессы, требующие немедленного ответа, могут иметь более высокий приоритет.

2. Очередь процессов:

Процессы сортируются в очереди планировщика в соответствии с их приоритетами. Процессы с высшим приоритетом будут находиться в верхней части очереди.

3. Выбор процесса для выполнения:

Планировщик выбирает процесс с наивысшим приоритетом из начала очереди для выполнения на процессоре.

4. Квант времени:

Процессу выделяется квант времени (time slice) для выполнения. Если процесс завершается или исчерпывает свой квант времени, планировщик выбирает следующий процесс в очереди.

5. Переключение контекста:

Когда происходит смена процессов, выполняется переключение контекста, при котором сохраняется состояние текущего процесса, и восстанавливается состояние следующего процесса в очереди

6. Динамическое изменение приоритета:

Во избежание голодания низкоприоритетных процессов, планировщик может динамически изменять приоритеты, повышая их по мере того, как процессы ожидают в очереди.

7. Обработка прерываний:

Если происходит прерывание, требующее немедленного внимания, планировщик может временно приостановить текущий процесс для обработки прерывания, после чего процесс либо продолжит выполнение, либо будет перемещен в очереди в соответствии с его приоритетом.

Этот механизм позволяет Linux эффективно обрабатывать множество процессов, обеспечивая справедливое использование процессорного времени и реагирование на срочные задачи, сохраняя при этом контроль над загрузкой системы и предотвращая голодание процессов.

28: Функционирование планировщика с разделением времени.

В операционной системе Linux планировщик с разделением времени используется для управления процессами, которым не требуется постоянный доступ к CPU, и обеспечивает справедливое распределение процессорного времени между ними. Вот как это работает:

1. Time Slicing:

- Каждому процессу в очереди назначается ограниченный интервал времени (time slice или quantum), в течение которого он может использовать CPU. Этот интервал времени обычно короткий, чтобы все активные процессы могли регулярно получать доступ к процессору.

2. Приоритеты и Квант времени:

- Хотя все процессы получают кванты времени, процессы с более высоким приоритетом могут получать их чаще или большие по длительности, в зависимости от реализации планировщика.

3. Циклическое Переключение:

- После истечения своего кванта времени, процесс помещается в конец своей очереди приоритета. Если есть другие процессы того же приоритета, следующий в очереди начнет выполняться.

4. Очереди:

- Процессы организованы в разные очереди в зависимости от их приоритета. Планировщик с разделением времени циклически перемещается по этим очередям, выделяя время каждому процессу поочерёдно.

5. Интервалы:

- Планировщик может прервать процесс, если он не завершил свою работу в отведенное время, и предоставить CPU следующему процессу в очереди. Это предотвращает доминирование одного процесса над всеми остальными и позволяет системе оставаться отзывчивой даже при запуске интенсивных задач.

6. Справедливость:

- Этот механизм гарантирует, что даже процессы с низким приоритетом будут иметь возможность выполнения, хотя и реже по сравнению с более высокоприоритетными процессами.

7. Динамическая Регулировка:

- Планировщик может динамически корректировать длину квантов времени в ответ на нагрузку системы и требования процессов, чтобы оптимизировать общую производительность и время отклика.

Планировщик с разделением времени в Linux обеспечивает, что система может обрабатывать множество процессов эффективно, без излишней задержки для пользователей и приложений, и поддерживает общую производительность системы на должном уровне, управляя временем CPU среди всех процессов.

29. Управление исполнением потока: диаграмма состояний потока. Управление состояниями.

Управление потоками в Linux включает в себя несколько состояний, в которых может находиться поток. Эти состояния отражаются в атрибуте `taskState` объекта задачи (task). Состояния и управление ими:

1. S (sleeping): Прерываемое ожидание. Поток ожидает события или ресурса. Может быть разбужен функцией `wakeup()` или другими потоками.

2. D (uninterruptible sleep): Непрерываемое ожидание. Поток ожидает события, но не может быть разбужен до тех пор, пока ожидаемое событие не произойдёт.

3. R (running или runnable): Готов к выполнению или выполняется. В этом состоянии поток может быть запланирован на выполнение планировщиком.

4. T (stopped): Процесс остановлен. Это может произойти в результате получения сигнала SIGSTOP. Процесс может быть продолжен сигналом SIGCONT.

5. Z (zombie): Поток завершил своё выполнение, но информация о нём сохраняется в системе до тех пор, пока родительский процесс не обработает его завершение.

Управление состояниями:

- Переход из состояния S в R происходит, когда функция `wakeup()` вызывается другим потоком или событием в системе (2 на диаграмме).

- Поток может перейти из состояния R в T, если он получает сигнал SIGSTOP (3 на диаграмме). Для возобновления работы потока, остановленного сигналом SIGSTOP, необходим сигнал SIGCONT, который переводит поток обратно в состояние R (4 на диаграмме).

- Когда поток выполняет системный вызов `exit()` или получает сигнал SIGTERM, он переходит в состояние Z (5 на диаграмме), становясь зомби. В этом состоянии он ждёт, пока его родительский процесс не обработает его завершение, что необходимо для освобождения ресурсов, связанных с потоком.

Эти состояния и переходы между ними являются частью жизненного цикла потока в многозадачной операционной системе. Планировщик ОС управляет этими переходами, обеспечивая эффективное распределение процессорного времени и ресурсов между всеми процессами в системе.

30. Изоляция программного кода. Защита системы. Режим исполнения user/supervisor и переключение между ними. Защита памяти. Защита устройств IO.

Изоляция программного кода и защита системы в операционных системах, таких как Linux, обеспечиваются через ряд механизмов:

1. Режимы исполнения (user/supervisor):

- В современных операционных системах существуют как минимум два режима исполнения: пользовательский (user mode) и режим супервизора (supervisor mode), который также называется режимом ядра (kernel mode).

- В пользовательском режиме код выполняется с ограниченными привилегиями, что предотвращает прямой доступ к аппаратным средствам и критическим частям системы.

- В режиме супервизора код выполняется с полным доступом к аппаратным средствам и может взаимодействовать с низкоуровневыми функциями системы.

2. Переключение между режимами:

- Программы обычно запускаются в пользовательском режиме и переключаются в режим супервизора через системные вызовы (syscalls) при необходимости выполнения привилегированных операций.

- После выполнения системного вызова происходит возврат в пользовательский режим.

3. Защита памяти:

- Современные операционные системы используют механизм виртуальной памяти для изоляции адресных пространств процессов. Каждому процессу предоставляется иллюзия собственного адресного пространства.

- Страницы памяти могут быть защищены правами доступа (например, только для чтения), что предотвращает несанкционированное изменение данных.

- Аппаратные средства, такие как MMU (Memory Management Unit), помогают контролировать доступ к физической памяти и обеспечивают, чтобы процессы не могли влиять друг на друга.

4. Защита устройств ввода-вывода (IO):

- Доступ к устройствам ввода-вывода строго контролируется ядром. Программы в пользовательском режиме не могут напрямую взаимодействовать с аппаратным обеспечением.

- Драйверы устройств, выполняющиеся в режиме супервизора, обеспечивают безопасный интерфейс между программным обеспечением и аппаратными средствами, обрабатывая запросы на ввод-вывод и предотвращая несанкционированный доступ.

Эти механизмы обеспечивают защиту и стабильность операционной системы, предотвращая случайные или злонамеренные попытки повреждения системы или компрометации данных. Они также важны для поддержания многозадачности и изоляции процессов, что является фундаментальным аспектом безопасности и стабильности в операционных системах.

31. Понятие системного вызова. Назначение и роль в ОС. Пространство (окружение) пользователя и ядра.

Системный вызов (syscall) — это программный интерфейс между пространством пользователя и пространством ядра операционной системы. Он позволяет программам в пользовательском пространстве запросить у ядра выполнение определённых функций, которые обычно связаны с низкоуровневым управлением ресурсами системы.

Назначение системных вызовов:

- Системные вызовы предоставляют стандартизированный способ выполнения операций, которые не могут быть выполнены напрямую программами в пользовательском пространстве из-за ограничений прав доступа. Это включает управление файлами, процессами, коммуникациями и другими важными функциями.

Роль системных вызовов в ОС:

- Системные вызовы являются мостом между приложениями пользователя и ядром ОС. Они позволяют приложениям выполнять действия с уровнем привилегий, который доступен только ядру, например, доступ к аппаратным устройствам или управление памятью.

- Они также обеспечивают контролируемое и безопасное взаимодействие, так как ядро ОС может валидировать запросы и данные, передаваемые через системные вызовы, что предотвращает непреднамеренные или злонамеренные действия.

Пространство (окружение) пользователя и ядра:

- Пространство пользователя — это область памяти, в которой выполняются приложения пользователя. Она изолирована от пространства ядра, что обеспечивает защиту данных и процессов системы от непреднамеренных ошибок или злонамеренных действий приложений.

- Пространство ядра — это защищённая область памяти, где выполняется код ядра ОС с полным доступом к аппаратным ресурсам и критически важным системным функциям.

Переключение между этими пространствами происходит при выполнении системного вызова, когда приложение в пользовательском пространстве запрашивает у ядра выполнение операции, которая требует привилегированного доступа. Это переключение защищено и контролируется операционной системой для поддержания безопасности и стабильности.

32. Процесс в ОС и его ресурсы: память, потоки, аргументы (cmdline), environment. Изоляция процессов. Создание процессов (fork())

Процесс в операционной системе — это экземпляр выполняемой программы. Он включает в себя код программы, её текущее состояние и контекст выполнения. Ключевые ресурсы, связанные с процессом, включают:

1. Память:

- Процессу выделяется отдельное адресное пространство, которое изолировано от других процессов. Это пространство включает в себя области для машинного кода (текст), данных, стека и кучи.

2. Потоки:

- В рамках процесса могут существовать потоки (threads), которые представляют собой "легковесные процессы", разделяющие адресное пространство и ресурсы процесса, но имеющие собственный стек вызовов и счётчик команд.

3. Аргументы (cmdline):

- Командная строка (cmdline), передаваемая процессу при его запуске, содержит аргументы, которые могут управлять поведением процесса.

4. Environment (окружение):

- Переменные окружения предоставляют контекст, в котором выполняется процесс, включая пути к библиотекам, настройки языка и другие конфигурационные данные.

Изоляция процессов:

- Операционная система использует механизмы защиты памяти, такие как виртуальная память и разделение прав доступа, чтобы изолировать процессы друг от друга и предотвратить случайное или умышленное воздействие процессов друг на друга.

Создание процессов (fork()):

- Системный вызов `fork()` используется в Unix-подобных системах для создания нового процесса путём "клонирования" текущего. Новый процесс называется дочерним, и он наследует копию адресного пространства родительского процесса. Это позволяет изолировать дочерний процесс от родительского, хотя они первоначально разделяют те же ресурсы.

- После `fork()`, обычно следует системный вызов `exec()`, чтобы загрузить и запустить новую программу в адресном пространстве дочернего процесса.

Эти механизмы и концепции обеспечивают основу для многозадачности и безопасности в современных операционных системах, позволяя процессам работать независимо и эффективно.

33. Запуск исполняемого модуля на выполнение в среде процесса (execve()). Загрузка динамических библиотек.

Системный вызов `execve()` используется в Unix-подобных операционных системах для запуска новой программы в адресном пространстве текущего процесса, заменяя его предыдущее содержимое. Вот основные моменты использования `execve()`:

1. Смена исполняемого кода процесса:

- При вызове `execve()` текущий исполняемый модуль процесса полностью заменяется новым. Счетчик команд, стек, данные и секции кучи перезаписываются данными из нового исполняемого файла.

2. Передача аргументов и переменных окружения:

- `execve()` принимает три аргумента: путь к исполняемому файлу, массив аргументов командной строки и массив переменных окружения. Эти параметры передаются новому процессу для его инициализации.

3. Загрузка динамических библиотек:

- После запуска новой программы операционная система автоматически занимается загрузкой динамических библиотек (.so файлов в Linux), которые требуются новому исполняемому модулю. Это делается с помощью динамического компоновщика (dynamic linker), который связывает библиотеки с программой во время выполнения.

Процесс загрузки динамических библиотек:

- Динамический компоновщик использует информацию из исполняемого файла о необходимых библиотеках (зависимостях).

- Библиотеки загружаются в память, если они уже не загружены. Если библиотеки уже присутствуют в памяти, используется их существующая копия (разделяемая память).

- Символы (функции и переменные) из этих библиотек связываются с адресами в памяти, чтобы программа могла их использовать.

Использование `execve()` и загрузка динамических библиотек являются важной частью процесса выполнения программы, обеспечивая, что все необходимые инструкции и ресурсы доступны процессу для его работы. Это также позволяет операционной системе оптимизировать использование памяти за счет разделения библиотек между несколькими процессами.

34. Аппаратный механизм виртуальной памяти (трансляции адресов – paging). Виртуальное и физическое адресное пространство. Содержимое таблиц страниц, PTE

Аппаратный механизм виртуальной памяти использует трансляцию адресов, известную как страничная организация памяти (paging), для преобразования виртуальных адресов, сгенерированных процессором, в физические адреса памяти.

1. Виртуальное адресное пространство:

- Виртуальное адресное пространство представляет собой непрерывный диапазон адресов, который операционная система предоставляет процессам. Оно позволяет каждому процессу работать с собственными адресами, как если бы он был единственным пользователем всей памяти.

2. Физическое адресное пространство:

- Физическое адресное пространство относится к реальной памяти — к тем адресам, которые физически существуют на устройствах хранения.

3. Таблицы страниц (page tables) и PTE (Page Table Entries):

- Каждый процесс имеет свою таблицу страниц, которая содержит записи (PTE), отображающие виртуальные страницы (фиксированного размера блоки виртуальной памяти) в соответствующие физические страницы.

- Содержимое PTE включает в себя физический базовый адрес страницы в памяти, а также флаги, указывающие на разрешения (чтение, запись, выполнение) и статус (присутствует в памяти, изменена, доступна).

Процесс трансляции адресов (paging):

- Когда процессор генерирует виртуальный адрес, система памяти разделяет его на части: номер страницы и смещение внутри страницы.

- Номер страницы используется для поиска в таблице страниц, чтобы найти соответствующую PTE.

- Физический адрес формируется путем объединения физического базового адреса, полученного из PTE, с исходным смещением.

Аппаратная поддержка:

- Современные процессоры имеют специализированные аппаратные блоки, такие как MMU (Memory Management Unit), которые автоматизируют процесс трансляции адресов и управления памятью.

- Кэширование таблиц страниц, например, с помощью TLB (Translation Lookaside Buffer), используется для ускорения процесса трансляции, поскольку доступ к физической памяти может быть относительно медленным.

Эти механизмы виртуальной памяти обеспечивают изоляцию между процессами, позволяют более эффективно использовать ограниченные ресурсы физической памяти и поддерживают необходимые уровни безопасности и стабильности в многозадачной операционной системе.

35. Организация таблиц страниц для архитектуры x86-64. Режимы адресации 48 бит и 57 бит.

В архитектуре x86-64 таблицы страниц организованы многоуровнево, что позволяет поддерживать большие объемы виртуальной и физической памяти.

Стандартный 48-битный режим адресации:

- Виртуальное адресное пространство 48 бит разделено на несколько уровней: PML4 (Page Map Level 4), PDPT (Page Directory Pointer Table), PD (Page Directory) и PT (Page Table).

- Каждая запись на каждом уровне таблицы страниц указывает на базовый адрес следующей таблицы, за исключением последнего уровня (PT), где записи указывают на физические страницы.

- Виртуальный адрес разделяется на части, которые определяют индексы в каждой из таблиц: 9 бит для PML4, 9 бит для PDPT, 9 бит для PD и 9 бит для PT, а оставшиеся биты представляют собой смещение внутри страницы.

Расширенный 57-битный режим адресации (5-Level Paging):

- С расширением виртуального адресного пространства до 57 бит в x86-64 (введенное в последних версиях архитектуры) добавляется дополнительный уровень индексации.

- Этот пятый уровень называется PML5 (Page Map Level 5), что позволяет поддерживать гораздо большее адресное пространство.

- Разделение виртуального адреса остается подобным 48-битной адресации, но добавляется еще один уровень индексации с 9 битами для PML5, что позволяет адресовать более обширную память.

Общие принципы:

- Страницы могут быть разного размера, включая стандартные 4 КБ, а также большие страницы (2 МБ или 1 ГБ), что позволяет уменьшить нагрузку на таблицы страниц для больших блоков памяти.

- Каждая запись в таблице страниц (PTE) содержит флаги, управляющие разрешениями (например, чтение, запись, выполнение), а также флаги состояния (например, присутствие в памяти, грязная страница, доступ к которой был).

Многоуровневая структура таблиц страниц в x86-64 позволяет операционной системе эффективно управлять большими объемами памяти и обеспечивать необходимые уровни изоляции и защиты памяти для различных процессов. Расширение до 57 бит открывает возможности для поддержки еще большего объема виртуальной памяти в будущих приложениях и системах.

36. Буфер быстрого преобразования адресов (TLB): устройство, принцип действия, влияние на производительность

Устройство TLB:

- TLB — это кэш, который используется аппаратным обеспечением виртуальной памяти для сокращения времени доступа к физическим адресам памяти.

- Он хранит недавно использованные преобразования виртуальных адресов в физические, избегая необходимости каждый раз обращаться к таблицам страниц, что является более длительной операцией.

Принцип действия TLB:

- Когда процессор генерирует виртуальный адрес, он сначала проверяет TLB на предмет наличия соответствующего физического адреса.

- Если соответствующая запись (TLB hit) найдена, то адрес получается немедленно, и операция продолжается без дополнительных задержек.

- Если запись в TLB отсутствует (TLB miss), то процессор должен обратиться к таблицам страниц в памяти, чтобы найти нужный физический адрес и обновить TLB данными этой трансляции.

Влияние на производительность:

- TLB может значительно увеличивать производительность системы, поскольку доступ к основной памяти для поиска в таблицах страниц занимает значительно больше времени.

- Чем больше размер TLB и чем лучше его алгоритмы работы, тем меньше вероятность промахов TLB и, соответственно, тем выше общая производительность системы.

- Однако, когда происходит TLB miss, производительность может снижаться из-за дополнительных затрат времени на доступ к таблицам страниц и загрузку соответствующих записей в TLB.

Управление TLB:

- Современные операционные системы и процессоры могут использовать различные стратегии для управления содержимым TLB, такие как принудительная инвалидация записей при смене контекста или использование различных уровней TLB для разных размеров страниц.

TLB является критически важной частью архитектуры современных компьютеров, поскольку он напрямую влияет на задержки доступа к памяти и общую производительность системы. Поэтому разработчики аппаратного обеспечения и операционных систем стремятся оптимизировать его работу для достижения наилучшей возможной эффективности.

37. Управление памятью в ОС с помощью paging: страничные рамки, таблицы страниц процессов. Варианты отображения страниц.

Страничные рамки и таблицы страниц процессов:

- Каждый процесс использует свою таблицу страниц, которая содержит Page Table Entries (PTE) для отображения виртуальных адресов на физические адреса в страничных рамках (page frames).

- ОС переключает таблицы страниц при переключении между процессами для обеспечения корректного отображения памяти для активного процесса.

- В модуле страниц заполнены только те PTE, которые имеют признак "present" (P=1), указывающий на наличие соответствующей страницы в физической памяти. Попытка доступа к странице, помеченной как несуществующая (P=0), приведет к исключению "page fault", которое обрабатывается диспетчером виртуальной памяти ОС.

Варианты отображения страниц:

- Private Page: PTE содержит номер page frame, выделенной данному процессу, и недоступной другим процессам.

- Shared Page: Несколько PTE, возможно в разных процессах, указывают на одну и ту же page frame, позволяя данным этой страницы быть доступными нескольким процессам.

- Absent Page: В PTE бит P=0, и обращение к такой странице вызывает page fault, но диспетчер памяти может модифицировать PTE или даже возвратить управление в прерванный процесс с измененным состоянием.

Управление исключениями page fault:

- Missing Page: Страница отсутствует в адресном пространстве, и это фиксируется как ошибка обращения к несуществующей памяти (SIGSEGV).

- Lazy Allocation: ОС может откладывать фактическое выделение page frame под выделенный процессу диапазон памяти до первого обращения к нему.

- Copy on Write (CoW): При `fork()` исходный набор page frames становится shared между новым и старым процессами, но в PTE обоих процессов сбрасываются биты R/W, и при попытке записи в такую страницу ОС создает ее private копию, исправляя PTE.

- Page on Disk: Страницы виртуального адресного пространства могут использоваться для отображения данных в файлах: memory mapped files.

Flags строки таблицы страниц (PTE Flags):

- R/W (Read/Write): Разрешение на чтение и запись в страницу.

- U/S (User/Supervisor): Разрешение на доступ к странице в режиме пользователя или супервизора.

- NX (No Execute): Запрет на выполнение кода на странице.

- A (Access): Был зафиксирован доступ к странице.

- D (Dirty): Была зафиксирована запись в страницу.

Использование этих механизмов позволяет ОС эффективно управлять памятью, предоставляя необходимую гибкость и контроль над распределением ресурсов между процессами и поддерживая защиту памяти и оптимизацию производительности.

38. Отображение файлов в память. Функция mmap()

Для отображения файлов в память используется функция `mmap()`, которая позволяет программному коду отобразить содержимое файла непосредственно в адресное пространство процесса. Этот процесс включает следующие шаги:

1. Резервирование адресного пространства:

- Диспетчер памяти ОС резервирует в виртуальном адресном пространстве процесса "окно" из нескольких страниц, где изначально страницы не заполнены, то есть PTE имеют признак "не присутствует" (P=0).

2. Обращение к страницам и page frames:

- При первом обращении к этим страницам происходит page fault, в ответ на который ОС выделяет соответствующие страничные рамки (page frames), которые считываются из соответствующих блоков файла.

3. Работа с отображенными данными:

- Код теперь может обращаться к данным файла, как к обычным содержимым памяти. Изменения, внесенные программным кодом в эти области памяти, синхронизируются с файлом в фоновом режиме.

4. Совместное использование отображенных данных:

- Одна и та же копия блоков файла может одновременно использоваться несколькими процессами, и при этом вносимые изменения будут видны всем процессам.

Функция `mmap()` обеспечивает эффективный механизм работы с файлами, позволяя избежать копирования данных между пространством пользователя и пространством ядра, что положительно сказывается на производительности приложений. Этот механизм широко используется для создания разделяемых памятных областей и для реализации memory-mapped файлов, что позволяет программам работать с большими файлами так, как если бы они были частью оперативной памяти.

39. Вытеснение неиспользуемых страниц на диск (swapping)

Для эффективного управления физической памятью операционная система использует механизм swapping, который позволяет вытеснять редко используемые страницы памяти в специально выделенный файл на диске, известный как swap file или файл подкачки.

Принцип работы swapping:

- С помощью бита A (Access) в PTE, диспетчер памяти отслеживает, к каким страницам процессов было обращение. Страницы, к которым обращения не было длительное время, могут быть вытеснены на диск.

- Если PTE страницы отмечено как отсутствующее (P=0), содержимое соответствующих ей страничных рамок при необходимости может быть записано в файл подкачки. В PTE записывается координата копии на диске.

- При повторном обращении к вытесненной странице диспетчер памяти считывает данные из файла подкачки и помещает их в новую страничную рамку, модифицируя PTE.

Влияние swapping на функционирование системы:

- Эффективно увеличивает объем доступной памяти, позволяя ОС использовать комбинацию физической памяти и места на диске (RAM+SWAP) для размещения данных процессов.

- Если физической памяти достаточно, то ОС старается дольше держать страницы в page cache и отбирать их только при необходимости выделения новых страничных рамок.

- При достаточном объеме физической памяти и swap файла страницы, которые фактически не используются, становятся "невидимыми" для системы, что позволяет более эффективно использовать память.

- При недостаточном объеме физической памяти может возникать эффект page thrashing, когда страницы часто вытесняются и загружаются обратно, что ведет к снижению производительности.

Настройка и управление swapping:

- Диспетчер памяти содержит множество параметров настройки, и при необходимости swapping можно отключать полностью, особенно в системах с большим объемом физической памяти.

Использование механизма swapping позволяет ОС гибко управлять ограниченными ресурсами памяти, обеспечивая при этом непрерывную работу приложений даже при высоких требованиях к памяти.

40. Общее адресное пространство в Linux. Канонические и неканонические 64-битные адреса

В архитектуре x86-64, используемой в современных версиях Linux, адресное пространство представляет собой 64-битное пространство, которое теоретически может адресовать до 2^64 локаций памяти. Однако, из-за ограничений архитектуры и операционной системы, используется не весь диапазон адресов:

Канонические адреса:

- Канонические адреса - это адреса, которые соответствуют используемому адресному пространству в архитектуре x86-64. В текущих реализациях Linux поддерживается 48-битное адресное пространство, что означает, что верхние 16 бит либо все установлены в 0, либо все установлены в 1, что соответствует знаковому расширению 48-битного адреса до 64 бит. Это обеспечивает доступ к адресному пространству размером 256 ТБ (128 ТБ для пользовательского пространства и 128 ТБ для пространства ядра).

Неканонические адреса:

- Неканонические адреса - это адреса, в которых верхние 16 бит не соответствуют знаковому расширению остальных 48 бит. Доступ к таким адресам не разрешен, и попытка их использования приведет к возникновению исключения в процессоре, которое обычно обрабатывается как ошибка сегментации.

Общее адресное пространство:

- В Linux общее адресное пространство разделено на пользовательское пространство и пространство ядра. Пользовательское пространство (нижняя половина адресного пространства) доступно для приложений, в то время как пространство ядра (верхняя половина) зарезервировано для ядра операционной системы и не доступно пользовательским процессам.

Это разделение обеспечивает безопасность и изоляцию, так как приложения не могут напрямую доступать память ядра, что помогает предотвратить случайные или злонамеренные повреждения критически важных структур данных ОС.

41. Логическая модель ввода-вывода: буфер, операции ввода, вывода и управления

Логическая модель ввода-вывода (I/O) представляет процесс копирования данных между памятью и внешними устройствами, для чего также требуется API управления внешними устройствами. В данной модели используются следующие компоненты и операции:

1. Буфер:

  - Область памяти, выделенная для временного размещения данных ввода-вывода. Буферы могут быть как для чтения (read buffer), так и для записи (write buffer).

2. Ввод (Input):

  - Процесс копирования данных с внешнего устройства в буфер чтения системной памяти. Это делается через операцию чтения (read).

3. Вывод (Output):

  - Процесс копирования данных из буфера записи системной памяти на внешнее устройство. Это осуществляется через операцию записи (write).

4. Функции управления внешними устройствами:

  - Отличаются в зависимости от устройства, но типичные функции включают в себя:

    - Переключение режимов работы устройства.

    - Опрос текущего статуса устройства.

    - Отправка команд устройству.

    - Получение индикации о завершении команд или приходе новых данных.

В логической модели ввода-вывода особое внимание уделяется управлению потоком данных между памятью и устройствами, а также синхронизации операций и обработке состояний устройств. API ввода-вывода предоставляет абстракцию, позволяющую программам взаимодействовать с разнообразным оборудованием без необходимости знания всех деталей их работы.

42. Виды доступа к данным на внешних устройствах: последовательный, произвольный, посимвольный, блочный

Внешние устройства ввода-вывода могут поддерживать различные виды доступа к данным:

1. Устройства с последовательным доступом:

- Input: Такие устройства позволяют только чтение данных последовательно, как пример можно привести клавиатуру.

- Output: Это устройства, которые позволяют только последовательную запись данных, например, принтер или терминал.

2. Pipes:

- Представляют собой комбинацию двух (логических) устройств: отправителя (output) и получателя (input), соединенных друг с другом.

3. Duplex channels:

- Это комбинация из двух pipes с встречным направлением обмена данными.

4. Устройства с произвольным доступом:

- Обычно это устройства хранения данных, к которым доступ возможен не последовательно. Операция seek() позволяет перемещаться по данным произвольно, что позволяет программно управлять порядком чтения/записи данных.

По организации обмена данными устройства делятся на:

1. Устройства с посимвольным (побайтным) обменом:

- Данные между устройством и CPU/памятью копируются фрагментами разной длины в зависимости от выполняемых операций и наличия данных.

2. Устройства с блочным обменом:

- Типично это устройства хранения данных (например, HDD, SSD), где данные организованы в блоки (чаще всего размером в 512 байт, реже 4096 байт). Обмен с такими устройствами происходит путём копирования целых блоков данных.

Эти виды доступа определяют, каким образом операционная система и программное обеспечение взаимодействуют с устройствами ввода-вывода, и влияют на проектирование системных архитектур и алгоритмов обработки данных.

43. Логическая концепция файла, операции логического ввода-вывода: read(),write(),lseek()

Файл в операционных системах рассматривается как упорядоченный набор байтов, над которыми можно выполнять операции логического ввода/вывода. Основные операции включают:

1. read():

- `int read(int fd, void *buf, int count)`

- Эта функция копирует данные из файла, на который указывает файловый дескриптор `fd`, в буфер `buf`. Количество байтов, которые должны быть прочитаны, указывается в `count`. Функция возвращает количество реально прочитанных байтов.

2. write():

- `int write(int fd, const void *buf, int count)`

- Операция записывает данные из буфера `buf` в файл, связанный с файловым дескриптором `fd`. Параметр `count` указывает количество байтов для записи. Возвращает количество фактически записанных байтов.

3. lseek():

- `off_t lseek(int fd, off_t offset, int whence)`

- Функция изменяет текущую позицию файлового указателя для файлового дескриптора `fd` в соответствии со значением `offset` и параметром `whence`, который может быть `SEEK_SET` (установка относительно начала файла), `SEEK_CUR` (относительно текущей позиции) или `SEEK_END` (относительно конца файла). Возвращает новую позицию указателя в файле.

Для работы с файлом в программе необходимо сначала получить доступ к нему с помощью функции `open()`, которая создает файловый дескриптор (file handle) - логическую "головку чтения/записи", и позволяет далее управлять файлом через его дескриптор.

- `int fd = open(const char *pathname, int flags)`

- Функция `open()` принимает путь к файлу `pathname` и флаги `flags`, определяющие режим доступа и поведение при открытии файла.

Операции `read()` и `write()` автоматически изменяют позицию файлового указателя, так что последовательные вызовы будут читать или писать последующие участки файла. Операция `lseek()`, в отличие от них, не влияет на содержимое файла, а только изменяет позицию, с которой будет производиться следующая операция чтения или записи. Функция `write()` может изменять размер файла в случае записи за пределы текущего конца файла.

44. Организация пространства имён файлов. Каталоги. Файловые системы и их виды. Монтирование. Подход Everything is a file.

Пространство имён файлов в операционных системах организовано иерархически, что обеспечивает структурированное и удобное хранение файлов и каталогов. Вот основные концепции:

1. Каталоги:

- Это специальные файлы, которые содержат информацию о других файлах и каталогах. Каталоги образуют древовидную структуру, начиная от корневого каталога и расширяясь до подкаталогов и файлов в них.

2. Файловые системы:

- Файловые системы определяют методы организации и доступа к данным на носителях информации. Они управляют способом хранения и извлечения данных. Существуют различные виды файловых систем, например:

  - Традиционные файловые системы (например, FAT, NTFS для Windows, EXT для Linux).

  - Сетевые файловые системы (например, NFS).

  - Распределённые файловые системы (например, Ceph или HDFS в экосистеме Hadoop).

3. Монтирование:

- Монтирование - это процесс, при котором файловая система делается доступной для использования в определенной точке монтирования в иерархии каталогов. После монтирования файловая система становится частью общего дерева каталогов.

4. Подход Everything is a file:

- Это философия Unix и Unix-подобных систем, согласно которой большинство ввода/вывода в системе обрабатывается через интерфейсы файлов, включая не только обычные данные на диске, но и устройства и сетевые соединения. Это унифицирует и упрощает интерфейс программирования, так как программы могут использовать стандартные системные вызовы для чтения, записи и управления практически любыми ресурсами системы.

Таким образом, организация пространства имён файлов, использование каталогов и файловых систем, а также процесс монтирования важны для управления данными и ресурсами в компьютерной системе. Эти концепции позволяют пользователям и программам легко ориентироваться в данных и эффективно работать с ними.

45. inodes и наборы данных. Отличие dataset от именованного файла. Метаданные набора данных. Типы наборов данных: regular, directory, char dev, block dev...

В Unix-подобных файловых системах inode (индексный узел) — это структура данных, которая содержит информацию о файлах и каталогах, то есть метаданные. Каждый файл или каталог ассоциируется с inode, который описывает его атрибуты и местоположение данных на диске.

Отличие dataset от именованного файла:

- Dataset (набор данных) относится к самим данным файла и его inode, в то время как именованный файл — это ссылка в каталоге, которая связывает имя с inode. Таким образом, inode и данные вместе образуют dataset.

- Файл может иметь несколько имен (жесткие ссылки), но они все будут указывать на один и тот же dataset.

Метаданные набора данных:

- Метаданные, хранящиеся в inode, включают:

  - Права доступа (маску доступа).

  - Владельца и группу файла.

  - Размер файла.

  - Временные метки, такие как время создания, модификации и последнего доступа.

  - Ссылки на блоки данных на диске, где хранится содержимое файла.

Типы наборов данных:

1. Regular (обычный файл): Содержит данные пользователя, например текст, изображения, исполняемые файлы и так далее.

2. Directory (каталог): Содержит ссылки на другие файлы и каталоги. Каждая ссылка ассоциирует имя файла с его inode.

3. Char dev (символьное устройство): Представляет устройство, доступ к которому возможен посимвольно, например, клавиатура или мышь.

4. Block dev (блочное устройство): Представляет устройство, доступ к которому возможен блоками, что эффективно для устройств с большим объемом данных, таких как жесткие диски.

Эти концепции являются фундаментальными для понимания устройства файловых систем и организации данных на уровне операционной системы. Информация, содержащаяся в inode, позволяет ОС управлять файлами и устройствами на более низком уровне, обеспечивая доступ к различным ресурсам и операциям над ними.

46. Алгоритм поиска набора данных по имени (pathname). Hardlinks и их ограничения. Symlinks.

Алгоритм поиска набора данных по имени (pathname traversal):

- Когда операционная система выполняет поиск набора данных (dataset), определенного inode, по имени файла (pathname), она выполняет процедуру, известную как pathname traversal.

- Процесс начинается с корневого каталога и последовательно просматривает записи каталогов, чтобы найти запись, соответствующую каждому элементу пути.

- В записях каталога хранится номер inode, либо следующего каталога в пути, либо самого файла (или устройства, named pipe, symlink).

Hardlinks:

- Жесткая ссылка (hardlink) — это запись в каталоге, которая напрямую указывает на inode файла и не отличается от "оригинального" имени файла.

- Hardlinks могут существовать только в пределах одной файловой системы, где расположен dataset, так как они указывают на номера inode в пределах этой файловой системы.

- Нельзя создавать hardlink на каталоги, чтобы предотвратить возможность создания циклических ссылок, которые могут затруднить обход файловой системы.

Symlinks:

- Символическая ссылка (symlink) представляет собой специальный тип файла, который содержит путь к другому файлу или каталогу.

- Symlinks могут пересекать границы файловых систем, так как они содержат путь, а не прямую ссылку на inode.

- При обработке открытия файла, если ОС обнаруживает symlink, она выполняет дополнительный шаг разрешения пути (pathname resolution), указанного в symlink, что может включать повторное выполнение pathname traversal для пути, указанного в symlink.

Эти механизмы позволяют гибко управлять файлами и доступом к ним, предоставляя возможность иметь несколько имен для одного файла (через hardlinks) и создавать псевдонимы или ярлыки для файлов и каталогов, находящихся в других местах файловой системы или даже на других носителях (через symlinks).

47. Доступ к наборам данных из процессов: file descriptors, file handles. Стандартные FD0,1,2

В операционных системах Unix-подобных, таких как Linux, процессы взаимодействуют с файлами через абстракции, известные как файловые дескрипторы и файловые дескрипторы (file handles).

File Descriptors (FD):

- Файловый дескриптор (FD) — это неотрицательное число, которое уникально идентифицирует открытый файл или другой поток данных (например, сокет или канал) в контексте процесса.

- Операционная система создает FD для файла при его открытии и освобождает его при закрытии файла.

- FD используется для выполнения различных операций ввода-вывода с помощью системных вызовов, таких как `read()`, `write()`, `close()`, `lseek()` и других.

File Handles:

- Файловый дескриптор (file handle) — это абстракция, предоставляемая языками программирования или API, которая ссылается на внутренний файловый дескриптор или другую структуру данных, содержащую информацию для доступа к файлу.

- В некоторых системах, например в Windows, термин "file handle" может использоваться для обозначения аналога файлового дескриптора.

Стандартные файловые дескрипторы:

- Стандартные файловые дескрипторы — это предварительно определенные дескрипторы, которые автоматически открыты для каждого процесса:

  - FD 0 (stdin): Стандартный ввод, обычно связан с клавиатурой.

  - FD 1 (stdout): Стандартный вывод, обычно связан с консолью или терминалом.

  - FD 2 (stderr): Стандартный вывод ошибок, также обычно связан с консолью или терминалом и используется для вывода сообщений об ошибках или диагностики.

Эти стандартные дескрипторы позволяют процессам взаимодействовать с пользователем или другими процессами через универсальные потоки ввода и вывода, не зависящие от конкретных устройств. Это обеспечивает консистентность и универсальность в работе с данными и упрощает написание переносимого кода между различными системами.

48. Системные вызовы управления файлами: open(), fcntl(), ioctl(), fstat()

open():

- `int fd = open(const char *pathname, int flags)`

- Этот системный вызов используется для открытия файла, идентифицируемого по пути `pathname`, и создания файлового дескриптора `fd` с заданными режимами доступа и управления, указанными в `flags`. Некоторые из флагов включают:

- `O_RDONLY`: Только чтение.

- `O_WRONLY`: Только запись.

- `O_RDWR`: Чтение и запись.

- `O_CREATE`, `O_TRUNC`, `O_APPEND`, `O_TMPFILE`: Флаги, управляющие созданием файла и его поведением при открытии.

fcntl():

- `int fcntl(int fd, int cmd, ...)`

- Функция `fcntl()` предоставляет широкий спектр управления файловым дескриптором `fd`, включая копирование дескриптора, изменение его свойств, и управление флагами:

- `F_DUPFD`, `F_DUPFD_CLOEXEC`: Копирование файлового дескриптора.

- `F_GETFL`, `F_SETFL`: Получение и установка флагов файлового дескриптора.

- `F_GETOWN`, `F_SETOWN`, `F_GETSIG`, `F_SETSIG`: Управление сигналами и их владельцем.

- `F_NOTIFY`: Уведомление об изменениях файлов в каталоге.

ioctl():

- `int ioctl(int fd, unsigned long request, ...)`

- Функция `ioctl()` используется для управления устройствами или файлами на более низком уровне, чем `fcntl()`, и может включать операции, специфичные для устройств:

- `FS_IOC_GETFLAGS`, `FS_IOC_SETFLAGS`: Получение и установка флагов inode.

fstat():

- `int fstat(int fd, struct stat *statbuf)`

- Системный вызов `fstat()` используется для получения информации об inode файла, связанного с файловым дескриптором `fd`. Он заполняет структуру `statbuf` метаданными файла, включая размер, права доступа, временные метки и другую информацию.

Эти системные вызовы являются основными инструментами для управления файлами и устройствами в операционной системе. Они предоставляют интерфейсы для выполнения стандартных операций ввода-вывода, а также для выполнения специализированных управленческих и конфигурационных задач.

49. Организация блочного ввода-вывода. Использование block(page) cache при чтении и записи

Блочный ввод-вывод (block IO) — это обмен данными с устройствами внешней памяти, такими как жесткие диски или SSD, которые организованы в секторы фиксированной длины (например, 512 байт или 4 килобайта).

Ключевые аспекты блочного ввода-вывода:

- Чтение/запись: Операции чтения и записи выполняются по полным секторам или последовательностям полных секторов, что обеспечивает эффективный обмен данными между памятью и хранилищем.

- Logical Block Number (LBN): Каждому сектору на устройстве соответствует последовательный логический номер блока, который представляет собой индекс в таблице размещения данных на устройстве.

- Block (page) cache: При чтении и записи используется кеш блоков — набор страничных рамок (page cache), который действует как промежуточный буфер, сохраняющий содержимое недавних операций ввода-вывода. Кеш блоков позволяет уменьшить количество обращений к диску, храня данные в памяти для более быстрого доступа.

- Взаимодействие с приложением: При чтении (`read()`) приложение копирует требуемый фрагмент данных из кеша блоков в программный буфер. При записи (`write()`) приложение модифицирует содержимое буфера, и эти изменения затем могут быть записаны обратно в кеш блоков и, в конечном итоге, на устройство хранения данных.

Процесс чтения и записи:

- При чтении системный вызов `read()` сначала проверяет, находится ли требуемый блок данных в кеше блоков. Если да, данные передаются в приложение из кеша, минуя доступ к диску.

- При записи системный вызов `write()` сначала помещает данные в кеш блоков. Оттуда данные могут быть асинхронно записаны на диск, что позволяет приложению продолжать работу без ожидания завершения операции записи на физическое устройство.

Эти механизмы обеспечивают более высокую производительность за счет уменьшения прямых операций чтения и записи на диск и повышают общую эффективность ввода-вывода системы.

50. Структура и управление page cache: фоновая запись, освобождение страничных рамок, алгоритм LRU, swapping

Структура и управление page cache в операционных системах, таких как Linux, включает следующие ключевые аспекты:

Структура page cache:

- Page cache состоит из page frames (PF), которые являются частью основной памяти и используются для кэширования данных из файлов и блочных устройств.

- Некоторые page frames привязаны к логическим блокам (logical blocks, LB) файлов/inodes и содержат копии их данных, позволяя операциям ввода-вывода использовать данные непосредственно из памяти.

Фоновая запись (write-back):

- Фоновый процесс операционной системы регулярно сбрасывает (flush) содержимое изменённых (Dirty) page frames на устройства хранения данных, чтобы освободить их для других данных и поддерживать количество свободных (free) page frames в пределах сконфигурированных рамок.

- Для page frames, не имеющих отображения на файлы/inodes (anonymous PF), используется swap file для сохранения их данных.

Освобождение страничных рамок:

- ОС стремится поддерживать определённое количество свободных page frames и может освобождать неиспользуемые страницы по алгоритму Least Recently Used (LRU).

- Все занятые page frames встроены в двусвязный список и при фиксации использования перемещаются в начало этого списка.

- Page frames, не используемые недавно, перемещаются в хвост списка, где алгоритм reclaim может выбирать кандидатов на освобождение.

- Page frames, исключаемые из page cache, помечаются как Invalid (P=0), и становятся доступными для других целей.

Алгоритм LRU:

- Алгоритм LRU используется для определения того, какие page frames следует освободить, отдавая приоритет страницам, которые не использовались дольше всего.

Swapping:

- Процесс reclaim (kswapd) запускается, когда количество свободных page frames снижается ниже определённого порога.

- Процесс не блокирует систему и работает в фоновом режиме, перенося содержимое PF на диск в swap file.

- Если количество free PF становится меньше нижнего предела, начинается блокировка процессов, требующих выделения PF до тех пор, пока необходимое количество PF не будет освобождено.

Эти механизмы управления page cache обеспечивают эффективное использование оперативной памяти, уменьшая задержки при доступе к часто используемым данным и оптимизируя общую

производительность системы за счёт сокращения количества операций ввода-вывода с физическими устройствами хранения данных.

51. Влияние page cache на эффективность работы системы. Объективные признаки недостатка памяти. Состояние thrashing

Page cache играет ключевую роль в повышении эффективности работы системы за счет уменьшения времени ожидания завершения операций ввода-вывода и уменьшения общего количества операций ввода-вывода. В условиях достаточного объема памяти, page cache практически устраняет задержки I/O между процессами и файловой системой, что делает файловое взаимодействие очень эффективным.

Объективные признаки недостатка памяти:

- При недостаточном объеме памяти, процесс reclaim (освобождение памяти) начинает активно отбирать страницы, что ведет к увеличению количества страницных прерываний (page faults) и операций записи на диск, что может привести к состоянию thrashing.

Состояние thrashing:

- Thrashing — это состояние, при котором система проводит большую часть времени на замену страниц в памяти, а не на выполнение полезной работы. Это происходит, когда система пытается освободить память, но из-за интенсивной работы приложений требуемые данные снова запрашиваются, что приводит к постоянной замене страниц и снижению производительности.

Методы борьбы с потерей данных и уменьшения задержек:

- Write through (O_DIRECT): Осуществление записи непосредственно на устройство минуя cache, что полезно для приложений, требующих гарантии немедленной записи данных на диск.

- fsync(): Системный вызов, который заставляет ОС немедленно записать на устройство изменения, накопленные в cache. Этот вызов требует программирования и используется для обеспечения целостности данных.

Использование page cache позволяет системе максимально эффективно использовать доступную память, ускоряя работу приложений и предотвращая лишние операции записи на физические носители. Однако, при недостаточном объеме памяти, необходимо применять специальные техники для управления page cache и предотвращения состояния thrashing, чтобы поддерживать оптимальную работоспособность системы.

52. Разметка блочных устройств: тома (raw devices), partitioning MBR/GPT, boot record

Разметка блочных устройств включает в себя следующие аспекты:

1. Тома (raw devices):

- Raw devices или "сырые" блочные устройства — это низкоуровневый доступ к устройству хранения данных, который позволяет взаимодействовать с ним без использования файловой системы. Это может быть полезно для приложений баз данных, которым нужен контроль над тем, как именно данные размещаются на диске.

2. Partitioning MBR/GPT:

- Partitioning — это процесс разделения физического устройства хранения данных на логические секции, называемые разделами.

- MBR (Master Boot Record): Традиционная система разделения, которая содержит загрузочный запись и таблицу разделов на первом секторе диска. MBR ограничен четырьмя основными разделами и поддерживает диски размером до 2 ТБ.

- GPT (GUID Partition Table): Более современная система разделения, которая является частью стандарта UEFI. GPT поддерживает диски большого размера (более 2 ТБ) и большее количество разделов (до 128 в базовой конфигурации).

3. Boot Record:

- Загрузочная запись — это область на устройстве хранения данных, которая используется для запуска операционной системы. В MBR-разметке загрузочный код содержится в Master Boot Record, тогда как в GPT-разметке загрузочная информация распределена между защищёнными областями на диске.

Разметка блочных устройств позволяет операционной системе и BIOS/UEFI идентифицировать, как и где начинаются и заканчиваются различные файловые системы и разделы на устройстве хранения данных. Это также необходимо для инициализации загрузки операционной системы и обеспечения безопасности данных на устройстве.

53. Массивы носителей данных: JBOD/SPAN, RAID 0, 1, 5, 6, 10, 50, 60

Массивы носителей данных представляют собой различные методы объединения нескольких физических дисков в логические единицы для улучшения производительности и надежности хранения данных. Вот основные типы таких массивов:

1. JBOD/SPAN (Concatenation):

- JBOD (Just a Bunch Of Disks) или SPAN — это объединение нескольких дисков таким образом, что они видны как единый большой диск. Данные записываются последовательно с одного диска на другой.

2. RAID 0 (Striping):

- RAID 0 разделяет данные на блоки и равномерно распределяет их по всем дискам в массиве, что увеличивает скорость чтения и записи, поскольку операции могут выполняться параллельно. Однако в этом режиме отсутствует избыточность, и отказ одного диска приводит к потере данных на всех дисках.

3. RAID 1 (Mirroring):

- RAID 1 создает точную копию (зеркало) данных на двух и более дисках, обеспечивая высокую надежность за счет полной избыточности. Если один диск выходит из строя, данные остаются доступными на его зеркале.

4. RAID 5 (Distributed Parity):

- RAID 5 распределяет информацию о паритете (данные для восстановления) по всем дискам в массиве. Это позволяет выдержать отказ одного диска, поскольку утраченные данные могут быть восстановлены с использованием паритетной информации.

5. RAID 6 (Double Distributed Parity):

- RAID 6 подобен RAID 5, но использует два блока паритета на каждый диск в массиве, что позволяет выдержать одновременный отказ двух дисков.

6. RAID 10 (1+0):

- RAID 10 комбинирует принципы RAID 1 и RAID 0, предлагая одновременно зеркалирование и разделение на полосы. Это обеспечивает как высокую производительность, так и надежность.

7. RAID 50 (5+0):

- RAID 50 объединяет несколько массивов RAID 5 в один большой массив с использованием striping, как в RAID 0. Это обеспечивает улучшенную производительность и надежность по сравнению с одним RAID 5.

8. RAID 60 (6+0):

- RAID 60 сочетает в себе массивы RAID 6 с striping RAID 0, обеспечивая еще большую избыточность и надежность, а также повышенную производительность благодаря распределению данных по множеству дисков.

Эти технологии позволяют оптимизировать системы хранения данных, сбалансировав потребности в скорости доступа к данным, их безопасности и стоимости хранения. Выбор конкретного типа RAID зависит от конкретных требований к хранилищу данных.

54. Роль и принципы функционирования logical volume manager: PV, VG, LV, PE

Logical Volume Manager (LVM) — это система управления томами в Unix-подобных операционных системах, которая обеспечивает гибкое управление дисковым пространством.

Компоненты LVM:

1. PV (Physical Volume):

- Это физический том, который может быть представлен целым диском, его разделом или RAID массивом. PV является основным строительным блоком в LVM и представляет физическое устройство хранения.

2. VG (Volume Group):

- Это группа томов, которая объединяет один или несколько PV в одну управляемую сущность. VG позволяет распределять пространство между логическими томами и предоставляет гибкость в управлении имеющимися ресурсами.

3. LV (Logical Volume):

- Логический том представляет собой часть пространства в группе томов, которую можно использовать под файловую систему или как raw-device. LV может быть увеличен или уменьшен в размерах динамически, без потери данных и без необходимости размонтирования.

4. PE (Physical Extent):

- Физический экстент — это единица разметки внутри PV, которая используется для управления распределением пространства. Каждый LV состоит из одного или нескольких PE, которые могут быть распределены по различным PV в рамках VG.

Функции LVM:

- Thin Provisioning:

- Это методика выделения дискового пространства, при которой физическое пространство выделяется не в момент создания тома, а по мере необходимости, что позволяет более эффективно использовать доступное дисковое пространство.

- Snapshots:

- Снимки состояния (snapshots) позволяют фиксировать состояние LV в определенный момент времени для создания резервных копий или восстановления данных.

Роль LVM заключается в предоставлении гибкости и управляемости дисковым пространством за счет абстрагирования физического хранения данных. LVM позволяет менять размеры томов, создавать снимки и производить множество других операций, не влияя на работу системы и приложений, что делает его ценным инструментом для администрирования систем.

55. Концепция дисковой файловой системы: superblock, управление пространством, управление наборами данных (файлами), управление именами

Концепция дисковой файловой системы включает несколько ключевых элементов, которые обеспечивают управление и организацию данных на диске:

1. Superblock:

- Superblock содержит метаданные всей файловой системы, такие как размер блока, общее количество блоков, количество свободных блоков, и так далее. Это своего рода "идентификационная карточка" файловой системы, необходимая для её функционирования.

2. Управление пространством:

- Файловая система управляет пространством через механизмы, такие как block groups и bitmap (битовые карты), которые помогают отслеживать, какие блоки заняты и какие свободны. Это позволяет оптимизировать использование дискового пространства и уменьшает фрагментацию.

3. Управление наборами данных (файлами):

- Управление файлами происходит через inode table и allocation map. Inode table содержит метаданные о каждом файле, включая права доступа, владельца, размер файла, и указатели на блоки данных. Allocation map отслеживает, какие блоки данных связаны с какими файлами.

4. Управление именами:

- Осуществляется через структуру каталогов (directories), которая связывает имена файлов с inode и позволяет организовать файлы в иерархическую структуру.

5. Metadata:

- Метаданные — это данные о данных, которые включают информацию о файлах и каталогах, такую как время создания и изменения, размер, права доступа и другие атрибуты.

6. Journal:

- Журналирование — это функция, которая повышает надежность файловой системы, записывая изменения в специальный журнал перед их применением к файловой системе. В случае сбоя это позволяет восстановить файловую систему в последнее консистентное состояние.

Эти элементы в совокупности образуют основу для организации и управления файловыми системами на блочных устройствах, таких как жесткие диски и твердотельные накопители, позволяя эффективно управлять хранением, поиском и доступом к данным.

56. Понятие параллелизма функционирования вычислительной системы и проблемы(задачи) вызванные ею

Параллелизм (concurrency) в вычислительных системах — это способность системы выполнять множество операций одновременно, что может включать выполнение инструкций различными ядрами CPU, операции ввода/вывода различными контроллерами IO, асинхронную передачу данных через сетевые адаптеры, отрисовку сцен графическим процессором (GPU), инициацию событий пользовательским интерфейсом (UI), и учет времени и заданных интервалов системными часами и таймерами.

Проблемы, вызванные параллелизмом:

- Ожидание событий: Вычислительная система должна уметь эффективно обрабатывать ожидание внешних событий, таких как таймеры, асинхронный ввод-вывод, реакция на действия пользователя и сетевые протоколы.

- Работа с разделяемыми данными (volatile): Необходимо управлять конкурентным доступом к данным, чтобы избежать гонок (race conditions) и обеспечить согласованность данных.

- Синхронизация процессов: Система должна синхронизировать процессы, включая ожидание завершения, потоковую обработку данных, модель взаимодействия издатель-потребитель (publisher-consumer).

- Фонки и критические секции: Необходимо обеспечить безопасность потоков (thread safety) при доступе к критическим секциям кода.

Механизмы ОС для решения этих проблем:

- Базовый механизм прерываний: Позволяет реагировать на события в реальном времени и обрабатывать их без ненужного ожидания.

- Диспетчеризация потоков: Распределяет время процессора между различными процессами и потоками, обеспечивая параллельное выполнение.

- Блокирующие и неблокирующие системные вызовы: Предоставляют способ ожидания событий без потери времени CPU (например, ожидание данных от диска).

- Примитивы синхронизации: Такие как мьютексы, семафоры, барьеры и условные переменные, которые обеспечивают синхронизацию и координацию между потоками.

Параллелизм позволяет системе быть более отзывчивой и эффективно использовать аппаратные ресурсы, но в то же время требует сложного управления и синхронизации для предотвращения ошибок в многопоточных и распределенных средах.

57. Ожидание в ОС: цикл ожидания, использование диспетчера/блокирующих системных вызовов.

В операционных системах механизмы ожидания используются для синхронизации выполнения задач, управления ресурсами и обработки асинхронных событий.

Циклы ожидания:

- Это методы, при которых процесс или поток регулярно проверяет наличие условия, необходимого для продолжения работы. Циклы ожидания могут быть неэффективны, поскольку занимают процессорное время, даже если условие для продолжения работы ещё не выполнено.

Блокирующий системный вызов:

- Процесс может войти в состояние ожидания через блокирующие системные вызовы, такие как `sleep()` или `usleep()`, которые приостанавливают выполнение на заданный интервал времени.

- Системные вызовы для ожидания сигналов, например `sigwaitinfo()` или `sigtimedwait()`, блокируют процесс до получения определённых сигналов.

- Для ожидания завершения других потоков используется вызов `wait`, который блокируется до тех пор, пока дочерний процесс не завершится.

Блокирующий ввод-вывод:

- Операции ввода-вывода также могут быть блокирующими, когда процесс ожидает завершения операции, например чтения данных с диска или сети.

Мультиплексирование ввода-вывода:

- Мультиплексоры, такие как `select()` или `poll()`, позволяют процессу ожидать события на нескольких файловых дескрипторах одновременно, улучшая эффективность ожидания и реакцию на события.

Таймеры:

- Системные таймеры, такие как `alarm()` и `timer_create()`, `timer_gettime()`, `timer_settime()`, позволяют процессам ожидать до истечения заданного времени, после чего может быть отправлен сигнал или вызван обработчик.

Очередь событий:

- Механизм, который управляет событиями в системе, позволяя процессам реагировать на изменения или запросы.

Диспетчеризация и блокирующие системные вызовы являются эффективными инструментами для управления ожиданием в многозадачной операционной системе. Они позволяют процессам освобождать процессорное время для других задач, когда они не могут продолжить выполнение

из-за отсутствия необходимых данных или событий. Это ключевой компонент синхронизации в параллельных и конкурентных вычислительных средах.

58. Таймеры в ОС. Алгоритмы управления аппаратными часами

В Linux таймеры и аппаратные часы используются для широкого спектра задач от планирования до управления энергопотреблением:

1. Таймеры в Linux:

- `alarm()` и `setitimer()`: Эти системные вызовы позволяют процессам запускать таймеры, которые посылают сигналы `SIGALRM` по истечении указанного времени.

- POSIX таймеры (`timer_create()`, `timer_settime()` и т.д.): Предоставляют более гранулярное и гибкое управление таймерами с возможностью связывания таймеров с потоками и установкой точных интервалов времени.

2. Управление аппаратными часами:

- Linux использует High Precision Event Timer (HPET) и другие аппаратные таймеры для точного измерения времени и планирования задач.

- Часы реального времени (RTC) используются для поддержания системного времени даже при выключенном компьютере.

3. Алгоритмы управления:

- Алгоритмы, такие как tickless kernel (бесшумное ядро), улучшают управление энергопотреблением, позволяя системе пребывать в спящем режиме дольше, когда нет активных задач.

- Компенсация дрейфа часов и синхронизация с NTP для обеспечения точности системного времени.

4. Встроенные Таймеры:

- Linux поддерживает также таймеры для отдельных драйверов и устройств, такие как watchdog timers, которые используются для обнаружения и восстановления после сбоев системы.

5. Планировщик и Таймеры:

- Планировщик задач Linux, CFS (Completely Fair Scheduler), использует таймеры для распределения процессорного времени между процессами, обеспечивая справедливое и эффективное планирование.

Linux предоставляет набор примитивов и механизмов для точного и гибкого управления временем и таймерами, что позволяет разработчикам и системным администраторам точно настраивать поведение системы в соответствии с их потребностями.

59. Проблема критической секции кода: примеры, гонки, решение на базе блокировок

В Linux, как и в других многозадачных операционных системах, проблема критической секции возникает, когда несколько потоков или процессов одновременно пытаются получить доступ к общему ресурсу, например, к общей переменной или структуре данных. Если доступ не синхронизирован, это может привести к состоянию гонки (race condition), когда результаты выполнения кода становятся непредсказуемыми и зависят от порядка, в котором инструкции выполняются потоками.

Примеры:

1. Два потока пишут в одну и ту же переменную: без защиты оба могут одновременно пытаться обновить переменную, что может привести к потере данных, если один поток перезапишет изменения, сделанные другим.

2. Потоки обновляют общий список: если два потока пытаются одновременно добавить элементы в список, это может привести к повреждению структуры данных, если одна из операций изменения нарушит связи между элементами.

Решение на базе блокировок:

Для решения проблемы критической секции в Linux обычно используются механизмы блокировок, такие как мьютексы (mutexes) и семафоры. Эти механизмы блокируют доступ к ресурсу, так что только один поток может войти в критическую секцию кода в данное время.

1. Мьютексы:

- Примитив синхронизации, который гарантирует, что только один поток может владеть мьютексом в данный момент времени. В Linux системный вызов `pthread_mutex_lock()` используется для захвата мьютекса перед входом в критическую секцию и `pthread_mutex_unlock()` для его освобождения после выхода.

2. Семафоры:

- Семафоры могут быть использованы для контроля доступа к ресурсу, которым могут пользоваться ограниченное количество потоков одновременно. В Linux семафоры управляются через API POSIX, с функциями `sem_wait()` для ожидания семафора и `sem_post()` для его освобождения.

3. Spinlocks:

- В ситуациях, когда ожидание блокировки должно быть очень коротким, можно использовать спинлоки, которые активно ожидают, вместо того чтобы усыплять поток. Они полезны в среде с высокой производительностью и низкой латентностью, но их использование должно быть осторожным, чтобы избежать избыточного потребления CPU.

Использование блокировок требует осторожности, чтобы избежать проблем, таких как взаимная блокировка (deadlock), когда два или более потоков ожидают освобождения ресурсов, заблокированных друг другом. В Linux разработчики должны тщательно проектировать многопоточные программы, чтобы обеспечить безопасную и эффективную синхронизацию.

60. Семафоры и их применение на примере producer-consumer

Семафоры в Linux — это примитивы синхронизации, которые используются для управления доступом к общим ресурсам и координации действий между потоками или процессами. Семафор может иметь значение, которое указывает количество единиц определённого ресурса, доступных для использования. Если значение семафора положительно, поток может занять ресурс, уменьшив значение семафора. Если значение семафора равно нулю, поток блокируется до тех пор, пока ресурс не станет доступен.

Примером применения семафоров является классическая проблема производителя и потребителя (producer-consumer), где производитель генерирует данные и помещает их в буфер, а потребитель извлекает данные из буфера. Если буфер пуст, потребитель должен ждать, пока производитель не положит в буфер новые данные. Если буфер полон, производитель должен ждать, пока потребитель не освободит место, извлекая данные.

В Linux реализация может выглядеть следующим образом:

1. Используются два семафора:

- Семафор `empty` инициализируется значением, равным размеру буфера, указывая на количество пустых мест.

- Семафор `full` инициализируется значением 0, указывая, что пока нет элементов для потребления.

2. Производитель (producer):

- Перед помещением элемента в буфер производитель делает операцию `wait` на семафоре `empty`. Это уменьшает значение семафора на 1.

- Если `empty` равен нулю, производитель блокируется, ожидая, пока потребитель не извлечет элемент из буфера.

- После помещения элемента в буфер производитель выполняет операцию `post` на семафоре `full`, уведомляя о наличии элемента для потребления.

3. Потребитель (consumer):

- Перед извлечением элемента из буфера потребитель делает операцию `wait` на семафоре `full`.

- Если в буфере нет элементов, потребитель блокируется до тех пор, пока производитель не добавит новый элемент.

- После извлечения элемента потребитель выполняет операцию `post` на семафоре `empty`, уведомляя о доступности свободного места в буфере.

Эти операции гарантируют, что производитель не будет переполнять буфер, и потребитель не будет извлекать данные из пустого буфера, предотвращая состояние гонки и другие проблемы синхронизации.

В Linux для работы с POSIX семафорами используются системные вызовы `sem_init()`, `sem_wait()`, `sem_post()` и `sem_destroy()`. Это позволяет реализовать механизмы синхронизации семафоров в пользовательском пространстве. Для работы с именованными семафорами, которые могут быть использованы между разными процессами, используются `sem_open()`, `sem_close()` и `sem_unlink()`.