Вопрос 1: Общее представление об операционных системах. Местоположение ОС. Функции ОС. Развитие операционных систем. Причины появления и существования ОС

Общее представление об операционных системах

Операционная система (ОС) — это комплекс программного обеспечения, управляющий аппаратными ресурсами компьютера и предоставляющий общие службы для компьютерных программ. Это посредник между пользователем/приложениями и аппаратным обеспечением.

Местоположение ОС

ОС располагается между аппаратным обеспечением (Hardware) и прикладным программным обеспечением (User Applications).

Иерархия:

1. **Пользователь/Приложения** (Браузеры, редакторы)
2. **Системные библиотеки/Утилиты** (API, оболочки)
3. **Операционная система (Ядро)**
4. **Аппаратное обеспечение** (CPU, RAM, Диски, Периферия)

Функции ОС

1. **Управление процессами**: создание, удаление, планирование исполнения, синхронизация и межпроцессное взаимодействие.
2. **Управление памятью**: распределение оперативной памяти, поддержка виртуальной памяти, защита памяти процессов.
3. **Управление файловой системой**: организация хранения данных, операции с файлами и директориями, разграничение доступа.
4. **Управление вводом-выводом**: взаимодействие с драйверами устройств, обработка прерываний, буферизация.
5. **Обеспечение безопасности**: аутентификация пользователей, контроль доступа к ресурсам, защита от сбоев.
6. **Сетевое взаимодействие**: поддержка сетевых протоколов и соединений.
7. **Предоставление интерфейсов**: API для программистов, CLI/GUI для пользователей.

Развитие операционных систем

1. **Нулевое поколение (1940-е)**: ОС отсутствуют, прямое программирование на машинном коде.
2. **Первое поколение (1950-е)**: Пакетная обработка (Batch systems), мониторные системы, последовательное выполнение задач (IBM 701).
3. **Второе поколение (1960-е)**: Мультипрограммирование, разделение времени (Time-sharing), многопользовательские системы (OS/360, Multics).
4. **Третье поколение (1970-1980-е)**: Миникомпьютеры, появление UNIX, стандартизация, развитие сетей (TCP/IP).
5. **Четвертое поколение (1980-е - настоящее время)**: Персональные компьютеры, графические интерфейсы (Windows, MacOS), распределенные системы, мобильные ОС.

Причины появления и существования ОС

1. **Управление сложностью оборудования**: Скрытие деталей аппаратной реализации от разработчика приложений (абстракция).
2. **Эффективность использования ресурсов**: Максимизация загрузки процессора и периферии (мультипрограммирование).
3. **Удобство использования**: Предоставление понятного интерфейса пользователю вместо работы с машинными кодами.
4. **Безопасность и изоляция**: Защита программ друг от друга и защита системы от сбоев одной программы.
5. **Стандартизация**: Обеспечение единого API для разработки ПО, переносимость программ.

Вопрос 2: Классификация ОС по функциональным характеристикам

1. По назначению

- **Универсальные (General-purpose)**: Решают широкий круг задач (Windows, Linux, macOS).
- **Специализированные**:
  - **Системы реального времени (RTOS)**: Гарантируют реакцию на события в заданные временные рамки (QNX, VxWorks). Делятся на *жесткие* (срыв сроков недопустим) и *мягкие* (срыв сроков снижает качество).
  - **Встроенные (Embedded)**: Для управления конкретным устройством, ограничены в ресурсах (Android, iOS - в широком смысле, TinyOS).

2. По режиму обработки задач

- **Пакетной обработки**: Задачи собираются в пакет и выполняются последовательно без участия пользователя.
- **Разделения времени (Time-sharing)**: Квантование времени процессора между несколькими задачами, создавая иллюзию одновременной работы.
- **Реального времени**: Приоритет скорости реакции на внешние события.

3. По количеству пользователей

- **Однопользовательские**: Обслуживают одного пользователя (MS-DOS, ранние Windows).
- **Многопользовательские**: Поддерживают одновременную работу нескольких пользователей с разграничением прав (UNIX, Linux, Windows Server).

4. По количеству задач

- **Однозадачные**: Одна программа в один момент времени (MS-DOS).
- **Многозадачные**: Параллельное (или псевдопараллельное) выполнение нескольких программ.
  - *Вытесняющая многозадачность*: ОС сама переключает процессы.
  - *Невытесняющая (кооперативная)*: Процесс сам отдает управление (старые версии Windows/MacOS).

5. По количеству процессоров

- **Однопроцессорные**: Работают на одном CPU.
- **Многопроцессорные**: Поддерживают распределение нагрузки между несколькими CPU/ядрами (симметричные SMP, асимметричные ASMP).

6. По типу использования ресурсов

- **Сетевые ОС**: Обеспечивают работу в сети, управление сетевыми ресурсами.
- **Распределенные ОС**: Управляют ресурсами нескольких компьютеров так, что они выглядят как единая система.

Вопрос 3: Структурная классификация ОС

Структурная классификация определяет организацию кода ядра и взаимодействие его компонентов.

Основные типы структур:

1. **Монолитная система**: Вся ОС выполнена как единая программа (ядро), работающая в привилегированном режиме. Компоненты (файловая система, драйверы, управление памятью) связаны напрямую.
2. **Слоистая (иерархическая) система**: ОС разбита на уровни. Каждый уровень пользуется сервисами только нижележащего уровня (Пример: THE, Multics).
3. **Микроядерная архитектура**: В привилегированном режиме работает только минимальное микроядро (планирование, IPC, базовая память). Остальные функции (драйверы, ФС) вынесены в пользовательские процессы-серверы.
4. **Модульная структура**: Ядро имеет базовый функционал, но может динамически подгружать модули (драйверы, ФС) во время работы (современный Linux, Solaris).
5. **Экзоядро**: Предоставляет безопасный доступ к аппаратуре с минимальными абстракциями, позволяя библиотекам приложений (libOS) реализовывать свои абстракции.
6. **Гибридная архитектура**: Сочетание подходов (обычно монолитного и микроядерного) для баланса производительности и структурированности (Windows NT, macOS).

Вопрос 4: Архитектуры ядер ОС. Монолитное ядро. Микроядро. Гибридное ядро

Монолитное ядро

Суть: Весь код ядра (планировщик, драйверы, ФС, сетевой стек) работает в едином адресном пространстве в режиме ядра (Kernel Mode). * **Взаимодействие**: Компоненты вызывают функции друг друга напрямую. * **Примеры**: Классический UNIX, Linux (с модульностью), MS-DOS, KolibriOS.

Микроядро

Суть: В режиме ядра работает только минимальный набор функций: 1. Межпроцессное взаимодействие (IPC). 2. Базовое управление памятью и планирование. 3. Обработка прерываний. Все остальные сервисы (драйверы, файловые системы,

сетевые протоколы) работают как обычные процессы в пространстве пользователя (User Mode). * **Взаимодействие**: Через обмен сообщениями (IPC). * **Примеры**: QNX, Minix, Mach, L4.

Гибридное ядро (Макроядро)

Суть: Компромисс между монолитным и микроядром. Большинство системных служб находятся в пространстве ядра для производительности, но структура стремится к модульности микроядра. * **Особенность**: Использует механизмы микроядер (IPC), но запускает ключевые компоненты (драйверы, стеки) в пространстве ядра, чтобы избежать накладных расходов на переключение контекста. * **Примеры**: Windows NT (XP, 7, 10, 11), XNU (macOS, iOS).

Вопрос 5: Достоинства и недостатки архитектур

Монолитное ядро

Достоинства: * **Производительность**: Высокая скорость работы за счет прямого вызова функций и отсутствия лишних переключений контекста. * **Зрелость**: Технологии хорошо отработаны.

Недостатки: * **Надежность**: Ошибка в любом драйвере может обрушить всю систему (BSOD / Kernel Panic). * **Сложность поддержки**: Огромный объем кода в одном месте, сложная отладка и модификация. * **Размер**: Занимает много памяти (если не модульное).

Микроядро

Достоинства: * **Надежность и устойчивость**: Падение драйвера (как отдельного процесса) не рушит ядро, его можно перезапустить. * **Расширяемость и гибкость**: Легко добавлять новые компоненты без изменения ядра. * **Безопасность**: Минимум кода с полными привилегиями.

Недостатки: * **Производительность**: Низкая из-за накладных расходов на переключение контекста и обмен сообщениями (IPC) между процессами-сервисами. * **Сложность разработки**: Требует сложной синхронизации и проектирования взаимодействий.

Гибридное ядро

Достоинства: * **Баланс**: Сочетает производительность монолита с структурированностью микроядра. * **Гибкость**: Удобная модель драйверов и подсистем.

Недостатки: * **Сложность**: Может наследовать недостатки обоих подходов (сложность отладки, уязвимость драйверов в режиме ядра).

Вопрос 6: Режимы работы ядра. Привилегированный и непривилегированный режимы

Режимы работы процессора

Для защиты ОС от действий пользователей и сбоев программ процессоры поддерживают как минимум два режима работы.

1. Привилегированный режим (Kernel Mode / Режим ядра / Ring 0)

- **Описание**: Режим с полным доступом ко всем аппаратным ресурсам и инструкциям процессора.
- **Кто работает**: Код ядра ОС, драйверы устройств.
- **Возможности**:
  - Выполнение любых инструкций CPU (включая управление прерываниями, переключение контекста).
  - Доступ к любой ячейке памяти.
  - Прямой доступ к портам ввода-вывода.

2. Непривилегированный режим (User Mode / Пользовательский режим / Ring 3)

- **Описание**: Режим с ограниченными правами для безопасного выполнения приложений.
- **Кто работает**: Прикладные программы (браузеры, игры), системные службы (в микроядрах).
- **Ограничения**:
  - Запрещено выполнение критических инструкций (остановка CPU, управление памятью).
  - Доступ только к выделенному адресному пространству.
  - Нет прямого доступа к оборудованию.

Переключение режимов (Context Switch)

Переход из User Mode в Kernel Mode происходит в строго определенных случаях:

1. **Системный вызов (System Call)**: Программа запрашивает сервис у ОС (открыть файл, выделить память).
2. **Прерывание (Interrupt)**: Сигнал от оборудования (нажатие клавиши, приход сетевого пакета).
3. **Исключение (Exception)**: Ошибка выполнения программы (деление на ноль, доступ к запрещенной памяти).

Обратный переход (Kernel -> User) происходит после обработки события командой возврата из прерывания.

Понятие процесса

Процесс — это абстракция ОС, представляющая собой экземпляр выполняемой программы. В отличие от программы (пассивного набора инструкций и данных на диске), процесс является активной сущностью, владеющей ресурсами: адресным пространством, открытыми файлами, сигналами и потоками исполнения. Процесс обеспечивает изоляцию выполнения программ друг от друга.

Вопрос 7: Память процесса. Инициализация и завершение процесса

Память процесса

Память процесса делится на следующие секции:

1. **Text (Code) Segment**: Содержит исполняемый код программы. Обычно доступен только для чтения и фиксирован по размеру.
2. **Data Segment**: Содержит инициализированные и неинициализированные (BSS) глобальные и статические переменные. Размер фиксирован.
3. **Heap (Куча)**: Динамически выделяемая память во время выполнения программы (malloc/new). Растет в сторону увеличения адресов.
4. **Stack (Стек)**: Содержит временные данные функций (локальные переменные, аргументы, адреса возврата). Растет в сторону уменьшения адресов.

Инициализация процесса

При создании процесса ОС выполняет следующие шаги:

1. **Загрузка**: Загрузка кода программы и данных из исполняемого файла в память.
2. **Выделение ресурсов**: Создание PCB, выделение памяти под стек и кучу.
3. **Инициализация контекста**: Установка регистров процессора (включая PC - Program Counter на точку входа) и переменных окружения.
4. **Состояние**: Перевод процесса в состояние *Ready*.

Завершение процесса

Процесс завершается либо добровольно (exit), либо принудительно (kill). 1. **Освобождение ресурсов**: Память, открытые файлы, дескрипторы освобождаются или закрываются. 2. **Статус завершения**: Код возврата передается родительскому процессу. 3. **Состояние Zombie**: Процесс переходит в состояние *Terminated* (Zombie), храня лишь минимальную информацию (PID, статус), пока родитель не считает её через `wait()`. После этого запись о процессе удаляется окончательно.

Вопрос 8: Отличие между процессами в Windows и Unix. Понятие Process Control Block (PCB), из чего он состоит

Отличия процессов Windows и Unix

- **Создание**:
  - *Unix*: `fork()` создает полную копию родительского процесса (с тем же кодом и данными). `exec()` заменяет код процесса на новый. Иерархия строго древовидная (init - корень).
  - *Windows*: `CreateProcess()` создает новый пустой процесс и загружает в него указанную программу. Иерархия процессов не поддерживается явно (нет строгого “родитель-потомок” владения, есть только хендлы).
- **Идентификация**: В Unix PID - число, в Windows - Handle (дескриптор) и PID.

Process Control Block (PCB)

PCB (Блок управления процессом) — это структура данных в ядре ОС, содержащая всю информацию, необходимую для управления конкретным процессом.

Состав PCB: 1. **Идентификатор процесса (PID)**: Уникальный номер. 2. **Состояние процесса**: (New, Ready, Running, Waiting, Terminated). 3. **Счетчик команд (Program Counter)**: Адрес следующей инструкции для выполнения. 4. **Регистры процессора**: Содержимое аккумуляторов, индексных регистров, стекового указателя и флагов (сохраняется при прерывании). 5. **Информация о планировании**: Приоритет, указатели на очереди планирования. 6. **Управление памятью**: Значения базового и граничного регистров, указатели на таблицы страниц/сегментов. 7. **Учетная информация**: Время процессора, реальное время работы, лимиты времени. 8. **Информация о вводе-выводе**: Список открытых файлов, выделенные устройства.

Вопрос 9: Состояния процесса. Таблица процессов. Моделирование режима многозадачности

Состояния процесса

Основные состояния жизненного цикла процесса:

1. **New (Создание)**: Процесс создается.
2. **Running (Выполнение)**: Инструкции выполняются на процессоре.
3. **Waiting (Ожидание/Blocked)**: Процесс ждет события (ввод-вывод, сигнал).
4. **Ready (Готовность)**: Процесс готов к выполнению, ждет освобождения процессора.
5. **Terminated (Завершение)**: Процесс закончил выполнение.

Переходы:

- *New -> Ready* (допуск к системе)
- *Ready -> Running* (планировщик выбрал процесс)
- *Running -> Ready* (истек квант времени или прерывание)
- *Running -> Waiting* (запрос ввода-вывода)
- *Waiting -> Ready* (завершение ввода-вывода)
- *Running -> Terminated* (выход)

Таблица процессов

Таблица процессов — системная структура данных, содержащая ссылки на PCB всех процессов (или сами PCB). Позволяет ОС быстро находить информацию о любом процессе.

Моделирование многозадачности

Многозадачность на одном процессоре реализуется через **псевдопараллелизм**:

1. Процессор выполняет процесс A.
2. По таймеру происходит прерывание.
3. ОС сохраняет контекст A в его PCB.
4. Планировщик выбирает процесс B.
5. ОС загружает контекст B из его PCB.
6. Процессор выполняет процесс B. Быстрое переключение создает иллюзию одновременной работы.

Вопрос 10: Механизм прерываний. Доступ пользовательских программ к функциям ОС. Контексты исполнения программного кода

Механизм прерываний

Прерывание — сигнал процессору о событии, требующем немедленного внимания. Приводит к приостановке текущего кода и вызову обработчика прерывания. * **Аппаратные**: От внешних устройств (таймер, диск, клавиатура). * **Программные (Исключения)**: Ошибки (деление на 0) или специальные инструкции (int, syscall).

Доступ к функциям ОС (Системные вызовы)

Пользовательские программы работают в ограниченном режиме (User Mode) и не могут напрямую обращаться к железу или памяти ядра. Доступ осуществляется через **Системные вызовы (System Calls)**:

1. Программа помещает аргументы в регистры.
2. Выполняет инструкцию прерывания (например, `syscall` или `int 0x80`).

3. Процессор переключается в режим ядра (Kernel Mode).
4. Ядро выполняет запрошенную операцию (например, `read`, `open`).
5. Ядро возвращает результат и переключает процессор обратно в режим пользователя.

Контексты исполнения

1. **Контекст процесса (User Context)**: Код, данные, стек пользователя, регистры при выполнении в User Mode.
2. **Контекст ядра (Kernel Context)**: Стек ядра, регистры, структуры данных ядра при выполнении системного вызова или обработчика прерывания в Kernel Mode.
3. **Контекст прерывания**: Специфическое состояние при обработке аппаратного прерывания (часто не связано с текущим процессом).

Вопрос 11: Кооперация процессов. Причины кооперации. Способы межпроцессной коммуникации

Кооперация процессов

Кооперация — взаимодействие процессов для выполнения общей задачи. **Причины**: 1. **Обмен информацией**: Несколько пользователей/процессов работают с одними данными. 2. **Ускорение вычислений**: Распараллеливание задачи на подзадачи (на многоядерных системах). 3. **Модульность**: Разделение системы на функциональные блоки (процессы). 4. **Удобство**: Выполнение нескольких задач одновременно (редактирование и печать).

Способы межпроцессной коммуникации (IPC)

1. **Общая память (Shared Memory)**: Процессы имеют доступ к общей области оперативной памяти. Самый быстрый способ, но требует синхронизации.
2. **Передача сообщений (Message Passing)**: Обмен пакетами данных через механизмы ОС (`send`/`receive`). Медленнее, но безопаснее (нет конфликтов доступа к памяти).
3. **Каналы (Pipes)**: Потоковая передача данных (см. вопрос 12).
4. **Сокеты (Sockets)**: Обмен данными между процессами на разных машинах (или одной) через сеть.
5. **Сигналы**: Простые уведомления о событиях.

Вопрос 12: Каналы (pipes). Особенности работы различных видов каналов. Организация процессов в группы

Каналы (Pipes)

Канал — это однонаправленный буфер для передачи потока байтов от одного процесса к другому.

Виды каналов

1. **Неименованные каналы (Anonymous pipes)**:
   - Создаются системным вызовом `pipe()`.
   - Существуют только в оперативной памяти пока открыты дескрипторы.
   - Доступны только родственным процессам (родитель-потомок), унаследовавшим дескрипторы.
   - Данные читаются в порядке FIFO.
2. **Именованные каналы (Named pipes / FIFO)**:
   - Существуют в виде специального файла в файловой системе.
   - Доступны любым процессам, знающим путь к файлу (даже неродственным).
   - Живут до явного удаления файла.

Организация процессов в группы

Группа процессов — объединение одного или нескольких процессов (обычно связанных одной задачей, например, конвейер `ls | grep`). * Каждая группа имеет уникальный идентификатор **PGID**. * Группы используются для управления сигналами: сигнал, отправленный группе, доставляется всем процессам в ней (например, Ctrl+C в терминале останавливает все процессы текущей группы). * **Сессия** объединяет несколько групп процессов (активная группа + фоновые).

Классификация систем

Вопрос 13: Классификация систем по Флинну: SISD, SIMD, MISD и MIMD. Локальные и распределенные системы в контексте параллельных вычислений

Классификация Флинна

Классификация архитектур вычислительных систем, предложенная Майклом Флинном в 1966 году, основана на понятии потока (последовательности команд или данных). Выделяют 4 класса:

1. SISD (Single Instruction, Single Data)

Одиночный поток команд, одиночный поток данных. * **Описание**: Классическая архитектура фон Неймана. В каждый момент времени один процессор выполняет одну инструкцию над одним элементом данных. * **Особенности**: Параллелизм отсутствует (возможен только конвейерный параллелизм внутри CPU). * **Примеры**: Традиционные одноядерные ПК (Intel 8086, 80486), простые микроконтроллеры.

2. SIMD (Single Instruction, Multiple Data)

Одиночный поток команд, множественный поток данных. * **Описание**: Одна инструкция выполняется одновременно на множестве процессорных элементов, обрабатывая разные данные. * **Особенности**: Синхронная параллельная обработка. Эффективно для работы с векторами, матрицами, массивами. * **Примеры**: Векторные процессоры, графические процессоры (GPU), матричные процессоры, расширения CPU (MMX, SSE, AVX).

3. MISD (Multiple Instruction, Single Data)

Множественный поток команд, одиночный поток данных. * **Описание**: Несколько процессоров выполняют разные инструкции над одним и тем же потоком данных. * **Особенности**: Данные передаются от одного процессора к другому (конвейерная обработка на уровне процессоров). Самый редкий тип архитектуры. * **Примеры**: Специализированные криптографические системы, системы отказоустойчивости (где несколько процессоров дублируют вычисления для проверки ошибок), систолические массивы.

4. MIMD (Multiple Instruction, Multiple Data)

Множественный поток команд, множественный поток данных. * **Описание**: Несколько независимых процессоров выполняют различные инструкции над различными данными. * **Особенности**: Полный асинхронный параллелизм. Самый распространенный класс современных параллельных систем. * **Примеры**: Многоядерные процессоры (SMP), кластеры рабочих станций, суперкомпьютеры.

Локальные и распределенные системы в контексте параллельных вычислений

Параллельные системы (обычно класса MIMD) делятся на два основных типа в зависимости от организации памяти и взаимодействия:

1. Локальные системы (Системы с общей памятью / Tightly Coupled)

- **Архитектура**: Процессоры (ядра) расположены физически близко (в одном корпусе/на одной плате) и соединены высокоскоростной шиной.
- **Память**: **Shared Memory (Общая память)**. Единое глобальное адресное пространство, доступное всем процессорам. Изменения памяти одним процессором видны другим.
- **Взаимодействие**: Через чтение и запись общих переменных в памяти.
- **Синхронизация**: Примитивы ОС (мьютексы, семафоры, спинлоки).
- **Примеры**: SMP (Symmetric Multiprocessing) сервера, многоядерные CPU (Core i7, Ryzen).

2. Распределенные системы (Системы с распределенной памятью / Loosely Coupled)

- **Архитектура**: Состоит из множества автономных узлов (компьютеров), соединенных коммуникационной сетью (Ethernet, Infiniband).
- **Память**: **Distributed Memory (Распределенная память)**. Каждый узел имеет собственную локальную память, недоступную другим напрямую.
- **Взаимодействие**: **Message Passing (Передача сообщений)**. Обмен данными происходит через явную отправку и прием сообщений по сети (MPI, RPC, Sockets).
- **Синхронизация**: Обмен сообщениями, барьеры.
- **Примеры**: Кластеры (Beowulf), Grid-системы, облачные вычислительные среды.

Понятие потока

Поток (Thread, Нить) — это наименьшая единица планирования и исполнения в операционной системе. Поток существует в рамках процесса и делит с другими потоками этого процесса адресное пространство и системные ресурсы (файлы), но имеет собственный контекст выполнения: счетчик команд, регистры и стек. Потоки позволяют реализовать параллелизм внутри одного приложения.

Вопрос 14: Причины создания потоков. Реализация сервера для обработки запросов через однопоточный процессы и множество потоков

Причины создания потоков

Потоки (threads) были созданы для:

1. **Модульности**: Возможность разбить приложение на логические последовательности действий.
2. **Эффективности**: Потоки легче (дешевле) создавать и уничтожать, чем процессы. Переключение контекста между ними быстрее, так как не нужно переключать адресное пространство.
3. **Параллелизма**: Возможность выполнять задачи одновременно на многоядерных процессорах.
4. **Асинхронности**: Пока один поток ждет ввода-вывода (например, от диска или сети), другие могут продолжать вычисления.

Реализация сервера

Однопоточный процесс Сервер работает в бесконечном цикле в одном потоке:

1. Ожидает запрос (`accept`).
2. Получает запрос, обрабатывает его.
3. Отправляет ответ.
4. Возвращается к ожиданию следующего.

- **Проблема**: Последовательная обработка. Если обработка одного клиента долгая, все остальные ждут в очереди. Не используется многоядерность.

Множество потоков (Multithreaded Server) Сервер имеет главный поток (Dispatcher) и рабочие потоки (Workers):

1. Главный поток ожидает запрос (`accept`).
2. При получении запроса создает **новый поток** (или берет из пула) и передает ему соединение.
3. Главный поток сразу возвращается к ожиданию новых запросов.
4. Рабочий поток параллельно обрабатывает свой запрос и завершается.

- **Преимущество**: Параллельная обработка множества клиентов. Блокировка одного потока (I/O) не останавливает сервер.

Вопрос 15: Причины создания потоков. Объекты, относящиеся к процессам и потокам. Стратегии реализации потоков

Объекты процесса и потока

В многопоточной среде процесс служит контейнером ресурсов, а поток — единицей исполнения.

- **Объекты процесса (общие для всех потоков)**:
  - Адресное пространство (код, глобальные данные, куча).
  - Открытые файлы (дескрипторы).
  - Дочерние процессы.
  - Сигналы и обработчики сигналов.
  - Учетная информация (UID, GID).
- **Объекты потока (индивидуальные)**:
  - Счетчик команд (PC).
  - Регистры процессора.
  - Стек (локальные переменные, цепочка вызовов).
  - Состояние (готов, выполняется, блокирован).

Стратегии реализации потоков

1. Потоки уровня пользователя (User-Level Threads, ULT) Реализуются библиотекой в пространстве пользователя (например, ранние Java, Green Threads). Ядро ОС видит только один процесс. * **Плюсы**: Очень быстрое переключение (не нужен системный

вызов), свой планировщик. * **Минусы**: Блокирующий системный вызов одного потока блокирует весь процесс. Не используют многоядерность (ядро дает процессу только 1 квант времени).

2. Потоки уровня ядра (Kernel-Level Threads, KLT) Поддерживаются непосредственно ОС (Windows, Linux, macOS). * **Плюсы**: Истинный параллелизм на многоядерных CPU. Блокировка одного потока не влияет на другие. * **Минусы**: Переключение медленнее (требует перехода в режим ядра). Занимают ресурсы ядра.

3. Гибридная реализация (M:N) Мультиплексирование M пользовательских потоков на N потоков ядра. Сложна в реализации, сейчас используется редко (пример: Go goroutines).

Вопрос 16: Параллельное программирование. Fork-and-Join модель

Fork-and-Join — модель параллельного выполнения, основанная на динамическом создании подзадач. 1. **Fork (Разветвление)**: Родительский поток разделяет задачу на части и запускает дочерние потоки для их выполнения. 2. **Join (Слияние)**: Родительский поток ожидает завершения всех дочерних потоков, после чего объединяет их результаты и продолжает работу.

Пример: Параллельная сортировка слиянием или обработка массива частями. Эффективность ограничивается законом Амдала (последовательной частью программы).

Вопрос 17: Параллельное программирование. Понятие псевдопараллелизма

Псевдопараллелизм (Конкурентность) — режим выполнения на однопроцессорной (одноядерной) системе, при котором создается иллюзия одновременной работы нескольких потоков. * **Механизм**: Процессор быстро переключается между потоками (Time Sharing). * **Суть**: В любой конкретный момент времени выполняется только одна инструкция одного потока. * **Отличие от истинного параллелизма**: При истинном параллелизме (на многоядерных системах) инструкции разных потоков выполняются физически одновременно.

Вопрос 18: Параллельное программирование. Ошибки в параллельном программировании. Гонка данных

Ошибки параллельного программирования

1. **Гонка данных (Data Race / Race Condition)**.
2. **Взаимная блокировка (Deadlock)**: Два потока ждут ресурсы, захваченные друг другом.
3. **Голодание (Starvation)**: Поток не получает доступ к ресурсам из-за низкого приоритета.
4. **Livelock**: Потоки активны, меняют состояния, но не продвигаются вперед (бесполезная работа).

Гонка данных (Race Condition)

Ситуация, когда поведение программы зависит от случайного порядка выполнения потоков. Возникает при одновременном доступе нескольких потоков к общим данным, где минимум один доступ — запись. * **Пример**: Два потока делают x++. 1. Поток А читает x (5). 2. Поток Б читает x (5). 3. Поток А пишет (6). 4. Поток Б пишет (6). *Итог*: 6 вместо 7. Информация потеряна. * **Решение**: Использование механизмов синхронизации (мьютексы, семафоры, атомарные операции).

Метрики параллельных алгоритмов

Вопрос 19: Верхние оценки ускорения. Закон Мура. Закон Амдала. Расчет ускорения и эффективности

Основные метрики

1. Ускорение (Speedup) Определение: Отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма.

$$S_p = \frac{T_1}{T_p}$$

Где:

- T_1 — время выполнения на 1 процессоре.

- T_p — время выполнения на p процессорах.

Интерпретация: * $S_p = p$ — **Линейное (идеальное) ускорение**. * $S_p < p$ — **Сублинейное (реальное) ускорение**. * $S_p > p$ — **Суперлинейное ускорение** (редкий случай, обычно связан с эффектами кэша).

2. Эффективность (Efficiency) **Определение**: Отношение ускорения к количеству используемых процессоров. Показывает среднюю загрузку процессоров.

$$E_p = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p}$$

Интерпретация: * $E_p = 1$ (100%) — идеальная загрузка. * Обычно $0 < E_p < 1$.

Закон Мура (Moore's Law)

Формулировка: Количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца (изначально 12-18). **Следствие**: Производительность процессоров росла экспоненциально долгое время за счет увеличения тактовой частоты и плотности транзисторов. Сейчас рост частоты остановился (из-за тепловыделения), и закон Мура реализуется через увеличение количества ядер (параллелизм).

Закон Амдала (Amdahl's Law)

Определяет теоретический предел ускорения при распараллеливании задачи. **Формулировка**: Ускорение программы ограничено временем, необходимым для выполнения её последовательной части.

Формула:

$$S_p = \frac{1}{f + \frac{1-f}{p}}$$

Где:

- f — доля последовательного кода (который нельзя распараллелить), $0 \leq f \leq 1$.
- $1 - f$ — доля параллелизуемого кода.
- p — количество процессоров.

Предел при бесконечном числе процессоров ($p \to \infty$):

$$S_{max} = \frac{1}{f}$$

Пример: Если 10% программы ($f = 0.1$) выполняется последовательно, то максимальное ускорение, даже при миллиарде процессоров, не превысит 10 раз (1/0.1).

Верхние оценки ускорения

Реальное ускорение всегда ограничено двумя факторами:

1. **Аппаратным пределом**: $S_p \leq p$ (нельзя ускориться больше, чем количество вычислителей).
2. **Алгоритмическим пределом (по Амдалу)**: $S_p \leq \frac{1}{f}$.

Таким образом, верхняя оценка:

$$S_p \leq \min\left(p, \frac{1}{f}\right)$$

Пример расчета

Дано:

- Последовательное время (T_1) = 100 сек.
- Параллельное время на 4 процессорах (T_4) = 40 сек.

1. **Ускорение**: $S_4 = \frac{100}{40} = 2.5$
2. **Эффективность**: $E_4 = \frac{2.5}{4} = 0.625$ (62.5%)

Понятие критической области

Критическая область (Critical Section) — это участок кода процесса/потока, в котором происходит доступ к общим разделяемым ресурсам (переменным, структурам данных, файлам). Для обеспечения корректности работы программы одновременное нахождение более чем одного потока в критической секции, связанной с одним и тем же ресурсом, должно быть запрещено (взаимоисключение).

Вопрос 20: Взаимоисключение с активным ожиданием

Критическая область (Critical Section)

Участок кода, в котором процесс обращается к общим ресурсам (переменным, таблицам, файлам), которые могут быть изменены другими процессами. Одновременно в критической секции может находиться только один процесс.

Условия корректного решения: 1. **Взаимоисключение**: Если процесс P_1 находится в критической секции, никакой другой процесс не может в ней находиться. 2. **Прогресс**: Если в критической секции никого нет, а какие-то процессы хотят войти, то выбор следующего не может откладываться бесконечно. 3. **Ограниченное ожидание**: Должен быть предел времени ожидания входа в критическую секцию.

Активное ожидание (Busy Waiting)

Способ реализации взаимоисключения, при котором процесс в цикле проверяет условие входа, непрерывно потребляя процессорное время («спин-блокировка»).

Вопрос 21: Взаимоисключение с активным ожиданием (Аппаратная поддержка)

Аппаратные инструкции

Для реализации блокировок процессоры предоставляют атомарные инструкции:

1. **Test-and-Set (TSL)**: Атомарно читает значение переменной и устанавливает её в 1.
2. **Compare-and-Swap (CAS)**: Атомарно сравнивает значение ячейки памяти с ожидаемым, и если они равны, записывает новое значение.

Спин-мьютекс (Spinlock)

Простейший примитив синхронизации на основе активного ожидания:

```
void lock(int *lock_var) {
    while (test_and_set(lock_var) == 1); //
}
void unlock(int *lock_var) {
    *lock_var = 0;
}
```

- **Плюс**: Эффективен, если ожидание очень короткое (меньше времени переключения контекста).
- **Минус**: Тратит процессорное время впустую. Не подходит для однопроцессорных систем без вытеснения.

Вопрос 22: Алгоритм Петерсона (мьютекс на N потоков)

Алгоритм Петерсона (для 2 потоков)

Программное решение проблемы взаимного исключения без использования специальных атомарных инструкций. Использует две переменные: массив флагов готовности `flag[]` и переменную очереди `turn`. Процесс сообщает о желании войти (`flag[i] = true`) и уступает очередь (`turn = j`), затем ждет, пока соперник либо не хочет войти, либо очередь вернется к нему.

Расширение на N потоков (Фильтр Петерсона)

Обобщение алгоритма для N потоков. Представляет собой N-1 уровней (“комнат ожидания”). Чтобы пройти на следующий уровень, поток должен стать “последним пришедшим” на текущем уровне и ждать, пока кто-то другой не придет и не сменит его на посту “швейцара”. * Поток входит в критическую секцию, только пройдя все N-1 уровней. * Гарантирует отсутствие голодания и взаимной блокировки.

Вопрос 23: Приостановка и активизация. Семафор, мьютекс и условная переменная

Примитивы без активного ожидания

Вместо траты циклов CPU процесс блокируется (переходит в состояние Waiting), и ОС пробуждает его, когда ресурс свободен.

1. Семафор (Дейкстра)

Целочисленная переменная, доступ к которой осуществляется только через две атомарные операции: `P` (wait/down) и `V` (signal/up). * **Счетный семафор**: Значение $N \geq 0$. Используется для управления доступом к пулу из N ресурсов. * **Бинарный семафор**: Значение 0 или 1. Аналог мьютекса.

2. Мьютекс (Mutex)

Объект синхронизации, который может находиться в двух состояниях: захвачен и свободен. * Имеет владельца (тот, кто захватил, должен и освободить). * Используется исключительно для защиты критических секций.

3. Условная переменная (Condition Variable)

Механизм, позволяющий потокам ждать наступления определенного события (условия). Всегда используется в связке с мьютексом (для защиты проверки условия). * `wait(cv, mutex)`: Атомарно отпускает мьютекс и блокирует поток. * `signal(cv)`: Разблокирует один ожидающий поток.

Вопрос 24: Приостановка и активизация. Отличия условной переменной от семафора

1. **Назначение**:
   - *Семафор*: Управление доступом к ресурсам (счетчик). Сигнал не теряется (увеличивает счетчик), даже если никто не ждет.
   - *Условная переменная*: Ожидание события (состояния данных). Сигнал теряется, если никто не ждет (потоки должны проверять условие сами).
2. **Память о прошлом**: Семафор хранит состояние (счетчик), условная переменная — нет.
3. **Использование**: CV всегда требует внешнего мьютекса для защиты данных условия, семафор самодостаточен.

Вопрос 25: Атомарные переменные (атомики). CAS инструкция. Спин-мьютекс

Атомарные переменные

Переменные (обычно целые числа или указатели), операции над которыми (чтение, запись, инкремент, сложение) выполняются неделимо на аппаратном уровне. Гарантируют целостность данных без использования тяжелых блокировок ОС.

CAS (Compare-And-Swap)

Фундаментальная инструкция для построения lock-free алгоритмов. `CAS(p, old, new)`:

1. Читает значение по адресу `p`.
2. Если оно равно `old`, записывает `new`.
3. Возвращает true/false (успех/неудача) или старое значение. Все это происходит атомарно.

Спин-мьютекс (на основе CAS)

Циклическая попытка захватить блокировку через CAS. `while (!CAS(&lock, 0, 1)) { pause(); }` Эффективен для очень коротких критических секций на многоядерных системах, так как избегает дорогого переключения контекста в ядро.

Вопрос 26: Передача сообщений. Барьеры

Передача сообщений (Message Passing)

Способ взаимодействия процессов (особенно в распределенных системах), где нет общей памяти. * Операции: `send(dest, msg)` и `receive(src, msg)`. * Может быть блокирующей (синхронной) или неблокирующей (асинхронной). * Решает проблемы синхронизации (нет гонок данных), но медленнее общей памяти из-за копирования данных.

Барьеры (Barriers)

Примитив синхронизации для группы потоков. Ни один поток не может продолжить выполнение, пока **все** потоки группы не достигнут точки барьера. * Используется в параллельных вычислениях, разбитых на фазы (шаги). * Пример: Все потоки должны закончить чтение данных, прежде чем начать обработку.

Понятие взаимоблокировки (Deadlock)

Вопрос 27: Условия возникновения. Моделирование взаимоблокировок. Способы борьбы со взаимоблокировками.

Определение

Взаимоблокировка (Deadlock) — ситуация, при которой группа процессов блокирует друг друга, так как каждый процесс удерживает ресурс и ждет освобождения другого ресурса, занятого другим процессом из этой же группы.

Условия возникновения (Условия Коффмана)

Взаимоблокировка возможна только при одновременном выполнении 4 условий:

1. **Взаимное исключение (Mutual Exclusion)**: Ресурсы не могут использоваться совместно (доступ эксклюзивен).
2. **Удержание и ожидание (Hold and Wait)**: Процесс, удерживающий минимум один ресурс, может запрашивать новые ресурсы, занятые другими процессами.
3. **Отсутствие вытеснения (No Preemption)**: Ресурс не может быть отобран у процесса принудительно; он освобождается только добровольно.
4. **Циклическое ожидание (Circular Wait)**: Существует замкнутая цепь процессов $\{P_0, P_1, ..., P_n\}$, где P_0 ждет ресурс, занятый P_1, P_1 ждет ресурс P_2, ..., а P_n ждет ресурс P_0.

Моделирование взаимоблокировок

Для анализа используется **Граф распределения ресурсов (Resource Allocation Graph)**:

- **Вершины**: Процессы (P) и Ресурсы (R).
- **Ребра**:
  - $P_i \to R_j$ (Запрос): Процесс i просит ресурс j.
  - $R_j \to P_i$ (Владение): Ресурс j занят процессом i.

Признак дедлока: * Если в графе **нет циклов** — дедлока нет. * Если есть цикл и каждый ресурс имеет **один экземпляр** — дедлок **гарантирован**. * Если есть цикл, но у ресурсов несколько экземпляров — дедлок **возможен**.

Способы борьбы со взаимоблокировками

1. Пренебрежение (Страус-алгоритм)

- **Суть**: Игнорировать проблему, предполагая, что дедлоки случаются редко.
- **Применение**: Windows, Linux (для пользовательских процессов). Если процесс завис, пользователь убивает его вручную.

2. Предотвращение (Deadlock Prevention)

- **Суть**: Сделать невозможным выполнение хотя бы одного из 4 условий Коффмана.
- **Методы**:
  - *Против удержания и ожидания*: Требовать захвата всех ресурсов сразу перед началом работы. (Минус: неэффективное использование ресурсов).
  - *Против циклического ожидания*: Упорядочить ресурсы (дать номера) и требовать захвата строго по возрастанию номеров.

3. Избегание (Deadlock Avoidance)

- **Суть**: Система анализирует каждый запрос ресурса и одобряет его только если это не приведет к **небезопасному состоянию**.
- **Метод**: **Алгоритм банкира (Banker's Algorithm)**. Требует знания заранее максимального количества ресурсов, которые понадобятся каждому процессу.

4. Обнаружение и восстановление (Detection and Recovery)

- **Суть**: Позволить дедлокам случаться, периодически запускать алгоритм поиска циклов в графе ожидания и устранять их.
- **Восстановление**:
  - Принудительное завершение процесса (Kill process).
  - Откат процесса к контрольной точке (Rollback).
  - Отбор ресурсов (Preemption).

Управление памятью

Управление памятью (Memory Management) — это подсистема операционной системы, отвечающая за выделение оперативной памяти процессам, её освобождение, защиту памяти процессов друг от друга и организацию эффективного совместного использования ограниченного объема физической RAM.

Вопрос 28: Адресные пространства. Понятие свопинга. Общий принцип управления памятью ОС. Распределитель памяти ядра

Адресные пространства

Виртуальное адресное пространство — диапазон адресов, доступных процессу (от 0 до MAX). ОС отображает эти виртуальные адреса на физические через таблицы страниц. * **Изоляция**: Процесс не видит памяти других процессов. * **Защита**: Сегменты имеют права (R/W/X).

Свопинг (Swapping / Paging)

Механизм, при котором ОС временно перемещает неактивные страницы памяти из RAM на жесткий диск (в swap-файл или раздел), освобождая место для активных процессов. При обращении к выгруженной странице происходит Page Fault, и она загружается обратно.

Общий принцип управления памятью

ОС выполняет:

1. **Трекинг**: Знает, какие части памяти свободны, а какие заняты.
2. **Аллокация**: Выделяет память процессам по запросу (malloc/brk/mmap).
3. **Освобождение**: Возвращает память в пул свободной.
4. **Виртуализация**: Транслирует виртуальные адреса в физические (MMU).

Распределитель памяти ядра (Kernel Allocator)

Отвечает за выделение небольших объектов внутри ядра (дескрипторы, структуры). * **Buddy System (Система двойников)**: Память делится на блоки размером 2^k. При запросе блок делится пополам, пока не будет получен подходящий размер. Быстрое объединение при освобождении. * **Slab Allocator**: Кэширует объекты фиксированного размера (например, `task_struct`), чтобы избежать фрагментации и ускорить выделение.

Оценка аллокаторов

- **Внутренняя фрагментация**: Потери памяти внутри выделенного блока (запросили 10 байт, дали 16).
- **Внешняя фрагментация**: Свободная память есть, но она разбита на мелкие куски, и нельзя выделить большой блок.
- **Скорость**: Время выполнения malloc/free.

Способы отслеживания свободной памяти

1. **Битовые карты (Bitmaps)**: Память делится на блоки, каждому соответствует бит (0-свободно, 1-занято). Компактно, но медленный поиск.
2. **Связанные списки (Linked Lists)**: Список свободных и занятых сегментов. Каждый элемент хранит размер и статус. Гибко, но сложнее объединять блоки.

Вопрос 29: Адресные пространства. Memory mapping

Memory Mapping (mmap)

Механизм отображения файла или устройства в виртуальное адресное пространство процесса. * Вместо чтения/записи файла через системные вызовы (`read`/`write`), процесс работает с файлом как с массивом в памяти. * **Ленивая загрузка**: Страницы файла загружаются в RAM только при обращении к ним (Page Fault). * **Разделяемая память**: Если несколько процессов делают mmap одного файла с флагом `MAP_SHARED`, они видят изменения друг друга и используют одни и те же физические страницы (экономия RAM).

Вопрос 30: Адресные пространства. Загрузчик программ в ОС. Секции памяти программ

Секции памяти (Сегменты)

Типичное адресное пространство процесса (снизу вверх):

1. **Text (Code)**: Исполняемый код. Read-only, Execute.
2. **Data**: Инициализированные глобальные переменные (`int x = 5;`). RW.
3. **BSS**: Неинициализированные глобальные переменные (`int y;`). Заполняется нулями при запуске. Не занимает места в файле на диске.
4. **Heap (Куча)**: Динамическая память (`malloc`). Растет вверх.
5. **Memory Mapping Segment**: Загруженные библиотеки (.so / .dll), mmap-файлы.
6. **Stack (Стек)**: Локальные переменные, адреса возврата. Растет вниз.

Вопрос 31: Алгоритм загрузки программы (exec)

1. **Проверка прав**: ОС проверяет, есть ли право на выполнение файла.
2. **Чтение заголовка**: Чтение ELF/PE заголовка для определения типа файла и архитектуры.
3. **Очистка**: Освобождение адресного пространства текущего процесса.
4. **Создание сегментов**:
   - Создание виртуальных сегментов (Text, Data, BSS) на основе заголовков программы.
   - Файл программы отображается (mmap) в эти сегменты (ленивая загрузка).
5. **Настройка стека**: Выделение стека, копирование аргументов командной строки (`argv`) и переменных окружения (`envp`) в стек.
6. **Загрузка интерпретатора** (опционально): Если программа динамически слинкована, загружается динамический линковщик (`ld-linux.so`).
7. **Запуск**: Установка счетчика команд (PC/RIP) на точку входа (`_start`), которая вызывает `main()`.

Вопрос 32: Статическая и динамическая линковка. Достоинства и недостатки

Статическая линковка

Весь код библиотек копируется в исполняемый файл на этапе компиляции. * **Плюсы**: Автономность (не нужны зависимости), немного быстрее запуск (нет резолвинга символов). * **Минусы**: Большой размер файла, дублирование кода в памяти (если 10 программ используют libc, будет 10 копий), сложно обновлять библиотеки (нужна перекомпиляция).

Динамическая линковка

В файл помещаются только ссылки на библиотеки. Код подгружается при запуске. * **Плюсы**: Экономия диска и памяти (одна копия библиотеки в RAM для всех процессов), легкое обновление библиотек (security fixes). * **Минусы**: Dependency hell (нужны правильные версии .so/.dll), чуть медленнее запуск и работа (из-за косвенной адресации PLT/GOT).

Динамическая загрузка (dlopen)

Программа сама просит ОС загрузить библиотеку в произвольный момент и получить адрес функции (`dlsym`). Используется для плагинов.

Вопрос 33: Таблицы PLT и GOT

Механизм для вызова функций из динамических библиотек (где адреса неизвестны до запуска). * **GOT (Global Offset Table)**: Таблица в секции данных, где хранятся реальные (абсолютные) адреса функций. Заполняется динамическим загрузчиком при запуске (или при первом вызове - lazy binding). * **PLT (Procedure Linkage Table)**: Короткие заглушки в секции кода. Когда программа вызывает `printf`, она прыгает в `printf@plt`. Заглушка смотрит в GOT и прыгает по адресу оттуда. * При первом вызове: адрес в GOT указывает обратно в загрузчик, который ищет реальную функцию, пишет адрес в GOT и вызывает её. * При последующих: в GOT уже реальный адрес, происходит прямой переход.

Вопрос 34: Позиционно-независимый код (PIC)

PIC (Position Independent Code) — код, который может работать будучи загруженным по любому адресу в памяти. * Необходим для разделяемых библиотек (.so), так как разные процессы могут загружать одну и ту же библиотеку в разные адреса виртуальной памяти. * Использует **относительную адресацию** (относительно PC/RIP) вместо абсолютной. * Для доступа к глобальным данным использует GOT.

Вопрос 35: Общая память процессов для кооперации

Shared Memory — самый быстрый способ IPC (Inter-Process Communication). * ОС отображает одни и те же физические страницы памяти в виртуальные адресные пространства разных процессов. * Данные не копируются (Zero-copy). * **Проблема**: Гонка данных. Требуется синхронизация (семафоры, мьютексы), которые должны располагаться в этой же общей памяти или использоваться отдельно.

Страничная организация памяти

Страничная организация памяти (Paging) — это схема управления памятью, при которой виртуальное адресное пространство процесса и физическая память делятся на блоки фиксированного размера (страницы и фреймы). Это позволяет избежать внешней фрагментации и эффективно реализовать механизм виртуальной памяти.

Вопрос 36: Виртуальная память. Общие положения. Таблица страниц. TLB. Поддержка большого объема памяти

Виртуальная память

Механизм управления памятью, при котором процессу предоставляется изолированное и непрерывное адресное пространство (виртуальное), которое отображается ОС на физическую память (RAM). Позволяет использовать больше памяти, чем есть физически (за счет свопинга на диск).

Таблица страниц (Page Table)

Структура данных в оперативной памяти (хранится отдельно для каждого процесса), которая содержит отображение виртуальных адресов на физические. * Виртуальная память делится на блоки фиксированного размера — **страницы (pages)** (обычно 4 КБ). * Физическая память делится на блоки того же размера — **фреймы (frames)**. * Каждая запись таблицы страниц (PTE - Page Table Entry) хранит: * Номер физического фрейма. * Биты присутствия (P), записи (W), доступа (A), модификации (D) и прав доступа (User/Supervisor).

TLB (Translation Lookaside Buffer)

Аппаратный кэш в процессоре (MMU), хранящий последние использованные трансляции «виртуальный адрес -> физический адрес». * Ускоряет доступ к памяти: при обращении к адресу сначала проверяется TLB (быстро). * Если записи нет в TLB (TLB Miss), происходит “прогулка” по таблице страниц (медленно) и загрузка записи в TLB.

Поддержка большого объема памяти (Многоуровневые таблицы)

Для больших адресных пространств (64-бит) хранить одну плоскую таблицу невозможно. Используются иерархические таблицы:

1. **Двухуровневая** (x86): Каталог таблиц -> Таблица страниц -> Фрейм.
2. **Четырехуровневая** (x64): PML4 -> PDPT -> PD -> PT -> Фрейм. Это позволяет хранить в памяти только те части таблицы, которые реально используются процессом (разреженность).

Вопрос 37: Виртуальная память. Оптимизации при работе со страничной памятью, сегментирование

Оптимизации страничной памяти

1. **Инвертированная таблица страниц**: Вместо таблицы на каждый процесс создается одна глобальная таблица, где запись соответствует физическому фрейму (а не виртуальной странице). Экономит память под таблицы, но затрудняет поиск (требует хэширования).
2. **Huge Pages (Большие страницы)**: Использование страниц размером 2 МБ или 1 ГБ вместо 4 КБ. Уменьшает размер таблиц страниц и количество промахов TLB (одна запись TLB покрывает больший объем памяти).
3. **Copy-on-Write (CoW)**: При создании процесса (fork) память не копируется физически. Родитель и потомок читают одни и те же страницы. Копирование происходит только при попытке записи.

Сегментирование

Альтернативный или дополнительный механизм управления памятью. * Память делится на логические блоки разного размера — **сегменты** (код, данные, стек). * Адрес состоит из: : . * **Плюсы**: Логическое разделение, защита на уровне модулей, возможность совместного использования кода. * **Минусы**: Внешняя фрагментация (сложно найти место для сегмента нужного размера). * **В современных ОС**: Используется **плоская модель памяти** (сегменты CS, DS, SS указывают на все адресное пространство), а основную работу выполняет страничный механизм. Сегментация осталась рудиментарно (например, для TLS - Thread Local Storage через FS/GS регистры).

Файловые системы

Файловая система — это часть операционной системы, определяющая способ организации, хранения и именования данных на носителях информации. Она предоставляет абстракцию “файлов” и “каталогов” для удобного взаимодействия пользователя и программ с данными, скрывая детали физического размещения на диске.

Вопрос 38: Общие положения файловых систем. Понятие файла. Структура файла. Типы и характеристики файлов. Понятие Каталога. Операции над файлами и каталогами

Понятие файла и его структура

Файл — именованная область данных на носителе информации. Это абстракция, скрывающая детали физического хранения. **Структура файла**: 1. **Неструктурированная последовательность байтов** (современный стандарт, Unix/Windows). ОС не знает о содержимом, смысл придают прикладные программы. 2. **Последовательность записей** (фиксированной или переменной длины). Чтение/запись идет записями. 3. **Дерево записей** (индексированные файлы, мейнфреймы). Поиск по ключу.

Типы и характеристики

Типы файлов: * Регулярные файлы (текстовые, бинарные). * Каталоги (директории). * Специальные файлы устройств (блочные и символьные). * Каналы (pipes) и сокеты. * Символические ссылки.

Характеристики (Атрибуты/Метаданные): Имя, размер, время создания/модификации/доступа, владелец (UID), группа (GID), права доступа (RWX), флаги (скрытый, системный).

Каталог

Каталог — файл, содержащий список соответствий «имя файла — идентификатор файла (inode)». Обеспечивает иерархическую структуру ФС. * **Корневой каталог (/)**. * **Текущий каталог (.)**. * **Родительский каталог (..)**.

Операции

- **Над файлами**: Create, Delete, Open, Close, Read, Write, Seek, Get/Set Attributes.
- **Над каталогами**: Create, Delete, Opendir, Readdir, Rename, Link (создание ссылки).

Общие понятия: Структура файловой системы и Реализация файлов

Структура файловой системы

Структура файловой системы определяет, как данные организованы на диске. Обычно диск разбивается на разделы (partitions). В начале раздела располагается **Загрузочный блок** (Boot block), затем **Суперблок** (Superblock), содержащий параметры ФС. Далее следуют структуры для управления свободным местом (битовые карты или списки), область **i-узлов** (inode table) и область данных (data blocks), где хранятся файлы и каталоги.

Реализация файлов

Реализация файлов решает задачу отображения логического потока байтов файла на физические блоки диска. Основная цель — обеспечить быстрый доступ (последовательный и произвольный) и эффективное использование места. Основные методы реализации: 1. **Непрерывное размещение**: Файл занимает смежные блоки. 2. **Связанный список**: Блоки разбросаны, каждый ссылается на следующий. 3. **Индексированное размещение (i-nodes)**: Специальная структура хранит адреса всех блоков файла.

Вопрос 39: Структура файловой системы. Реализация файлов. Непрерывное размещение файлов и I-узлы. Совместно используемые файлы

Непрерывное размещение

Файл занимает последовательный набор блоков диска. * **Плюсы**: Простая реализация (хранится только начальный блок и длина), максимальная скорость чтения (нет поиска). * **Минусы**: Внешняя фрагментация диска, сложность расширения файла (если нет места сразу за ним). Применимо для CD/DVD (ISO 9660).

I-узлы (Index Nodes)

Каждому файлу соответствует структура данных — **i-node**, содержащая атрибуты файла и адреса его блоков. * **Адресация**: * Прямые адреса (на первые N блоков). * Косвенный адрес (на блок, содержащий адреса). * Двойной/тройной косвенный адрес (для больших файлов). * **Плюсы**: Быстрый доступ, отсутствие внешней фрагментации, легкое расширение файла.

Совместно используемые файлы (Hard Links)

Реализуются через **жесткие ссылки**. В разных каталогах (или одном) создаются записи с разными именами, указывающие на **один и тот же i-node**. В i-node ведется счетчик ссылок. Файл удаляется физически только когда счетчик равен 0.

Вопрос 40: Структура файловой системы. Реализация файлов. Жесткие и символические ссылки

Жесткие ссылки (Hard Links)

- Прямая ссылка на **i-node** файла.
- Все жесткие ссылки на файл равноправны (исходное имя — тоже жесткая ссылка).
- **Ограничения**: Нельзя ссылаться на каталоги (во избежание циклов), нельзя ссылаться на файлы в другой файловой системе (i-node уникален только в пределах одной ФС).

Символические ссылки (Symbolic/Soft Links)

- Специальный файл, содержащий **путь** к целевому файлу.
- Имеет свой собственный i-node.
- **Особенности**: Могут указывать на несуществующие файлы (“битые ссылки”), могут ссылаться на каталоги и файлы в других ФС. Медленнее жестких (нужен парсинг пути).

Вопрос 41: Структура файловой системы. Реализация файлов. Суперблок как описатель файловой системы

Суперблок

Ключевая структура данных, располагающаяся в начале раздела диска. Описывает параметры всей файловой системы. При монтировании считывается в память. **Содержит**: 1. Тип файловой системы (Magic number). 2. Общее количество блоков и i-узлов. 3. Количество свободных блоков и i-узлов. 4. Размер блока. 5. Указатели на списки свободных блоков/i-узлов (или битовые карты). 6. Флаги состояния (чистая/грязная размонтировка).

Повреждение суперблока делает ФС нечитаемой (поэтому хранятся его копии).

Вопрос 42: Структура файловой системы. Реализация файлов. Размещение с использованием связанного списка

Связанный список блоков

Файл хранится как цепочка блоков. В начале каждого блока несколько байт отводится под указатель на следующий блок. * **Плюсы**: Нет внешней фрагментации, файл может расти пока есть место. Метаданные (в каталоге) хранят только адрес первого блока. * **Минусы**: Очень медленный произвольный доступ (чтобы прочитать конец, нужно пройти весь список), данные блока не степень двойки (из-за указателя), ненадежность (потеря одного блока рвет цепочку).

Связанный список с таблицей (FAT)

Указатели вынесены из блоков данных в отдельную таблицу в памяти (**FAT - File Allocation Table**). Таблица индексируется номером блока. * `Table[i]` хранит номер следующего блока за блоком `i`. * **Плюсы**: Весь блок под данные, быстрый произвольный доступ (если таблица в RAM). * **Минусы**: Таблица может быть огромной для больших дисков, должна постоянно находиться в памяти.

Вопрос 43: Структура файловой системы. Реализация каталогов. Оптимизации при поиске в каталогах

Реализация каталогов

Каталог — это файл, содержащий список записей. Каждая запись: `(  ,  i-  )`. 1. **Линейный список**: Простой массив записей. Поиск требует линейного сканирования ($O(N)$). Удаление требует сдвига или пометки “пусто”. Медленно для больших каталогов. 2. **Хеш-таблица**: Используется хеш от имени файла для быстрого поиска ($O(1)$). Сложнее в реализации (коллизии). 3. **B-деревья**: Сбалансированные деревья (используются в NTFS, XFS, современных ext4). Гарантируют логарифмическое время поиска ($O(\log N)$) и сортировку имен.

Оптимизации поиска

- **Кэширование путей (dentry cache)**: В оперативной памяти (VFS) хранятся результаты трансляции путей в i-узлы, чтобы не читать диск каждый раз при обращении по пути.

Вопрос 44: Структура файловой системы. Виртуальные файловые системы

Виртуальная файловая система (VFS)

Абстрактный слой в ядре ОС, предоставляющий единый интерфейс (API) для работы с различными файловыми системами (ext4, NTFS, FAT32, NFS). * **Принцип**: Приложения используют системные вызовы `open`, `read`, `write`. VFS перенаправляет их к драйверу конкретной ФС. * **Объекты VFS**: * `superblock` (описание ФС). * `inode` (метаданные файла). * `dentry` (элемент пути/каталога, кэшируется). * `file` (открытый файл, связан с процессом).

Вопрос 45: Структура файловой системы. Оптимальный выбор размер размера блока. Принципы отслеживания пустых блоков

Выбор размера блока

Блок — минимальная единица обмена с диском и выделения места. * **Маленький блок** (0.5-1 КБ): Экономия места (малая внутренняя фрагментация для мелких файлов), но низкая скорость (много операций чтения/записи, большие таблицы размещения). * **Большой блок** (8-64 КБ): Высокая скорость (меньше операций для считывания объема данных), но большая потеря места (внутренняя фрагментация) для мелких файлов. * **Оптимум**: Обычно 4 КБ (компромисс + совпадает с размером страницы виртуальной памяти).

Отслеживание пустых блоков

1. **Связанный список свободных блоков**: В суперблоке ссылка на первый свободный, в нем — на следующий (или на группу свободных). Эффективно, если блоки не нужно искать подряд.
2. **Битовая карта (Bitmap)**: Массив битов, где бит `i` равен 1, если блок занят, и 0, если свободен. Компактно, легко находить непрерывные группы блоков, легко загружать в процессорный кэш.

Вопрос 46: Структура файловой системы. Поддержание непротиворечивости файловой системы. Увеличение производительности при работе с блоками файловой системы. Отображение файлов на оперативную память

Непротиворечивость (Consistency)

Сбои питания могут нарушить целостность ФС (файл записан, а запись в каталоге нет). 1. **Утилиты проверки (fsck)**: Сканируют весь диск, сверяют списки блоков и ссылки, исправляют ошибки. Очень медленно. 2. **Журналирование (Journaling)**: Перед изменением ФС запись о намерении (“транзакция”) пишется в журнал. После сбоя достаточно повторить операции из журнала. Быстрое восстановление.

Увеличение производительности

1. **Блочный кэш (Buffer Cache)**: Хранение часто используемых блоков диска в оперативной памяти.
2. **Упреждающее чтение (Read-ahead)**: При последовательном чтении ОС заранее грузит следующие блоки в кэш.
3. **Отложенная запись**: Данные пишутся в кэш, а на диск сбрасываются позже (группами), что снижает количество обращений к диску.

Отображение файлов (Memory Mapping, mmap)

Механизм, позволяющий отобразить файл (или его часть) в виртуальное адресное пространство процесса. * Файл становится доступен как массив байтов в памяти. * ОС подгружает страницы файла по требованию (page fault). * Исключает лишнее копирование данных (buffer cache -> user space).

Вопрос 47: Как работает Copy-On-Write

Copy-On-Write (Копирование при записи)

Оптимизационная стратегия управления ресурсами. * **Суть**: При запросе на копирование данных (например, `fork()` процесса) реального физического копирования не происходит. Оба потребителя (родитель и потомок) получают ссылки на одни и те же страницы памяти, помеченные как “только для чтения”. * **Срабатывание**: Если кто-то пытается **записать** данные, возникает исключение процессора (page fault). ОС перехватывает его, выделяет новую физическую страницу, копирует туда данные и разрешает запись для инициатора. * **Выгода**: Экономия памяти и времени, если данные не изменяются (или изменяются редко).

```
//
int fd = open("file.txt", O_RDWR);
void* addr = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
// Fork      CoW-
//
pid_t pid = fork();
//                    :
if (pid == 0) {
    *(char*)addr = 'X'; // CoW:
}
```

Сетевое программирование

Сетевое программирование — это создание программного обеспечения, компоненты которого взаимодействуют друг с другом через компьютерную сеть. Оно включает в себя использование сетевых протоколов (TCP/IP), адресацию и обмен данными между процессами на удаленных узлах.

Вопрос 48: Понятие сокета. Именное пространство в сети. Виды установления соединения и передачи данных. Порядок байтов, передаваемых по сети

Понятие сокета

Сокет — это программный интерфейс (абстрактный объект) для обеспечения информационного обмена между процессами. * Для приложения сокет — это файловый дескриптор, с которым можно работать функциями `read`/`write`. * Сокет определяется парой: **IP-адрес + номер порта**.

Именное пространство

Для адресации в сетях TCP/IP используется иерархия:

1. **MAC-адрес** (Канальный уровень): Физический адрес сетевой карты (48 бит). Используется внутри локальной сети.
2. **IP-адрес** (Сетевой уровень): Логический адрес узла в сети (IPv4 — 32 бита, IPv6 — 128 бит).
3. **Порт** (Транспортный уровень): Число (16 бит), идентифицирующее конкретный процесс/службу на хосте.

Виды установления соединения и передачи

По установлению соединения: * **С установлением соединения (Connection-oriented, TCP)**: Гарантирует создание логического канала перед передачей данных. * **Без установления соединения (Connectionless, UDP)**: Данные посылаются получателю без предварительной проверки его готовности.

По типу адресации: * **Unicast** (один-к-одному). * **Broadcast** (один-ко-всем в подсети). * **Multicast** (один-к-группе).

Порядок байтов

Разные архитектуры процессоров хранит многобайтовые числа по-разному:

- **Little-Endian** (Intel x86): Младший байт по младшему адресу.
- **Big-Endian** (Сетевой порядок, Network Byte Order): Старший байт по младшему адресу.

В заголовках пакетов IP/TCP/UDP всегда используется **Big-Endian**. Функции преобразования:

- `htons()` / `htonl()`: Host to Network (Short/Long).
- `ntohs()` / `ntohl()`: Network to Host (Short/Long).

Вопрос 49: Протоколы TCP и UDP. Достоинства и недостатки

TCP (Transmission Control Protocol)

Протокол управления передачей. Ориентирован на соединение, обеспечивает надежную доставку потока байтов. * **Достоинства**: * Гарантия доставки (автоматический повтор потерянных пакетов). * Сохранение порядка следования данных. * Контроль потока (Flow Control) и перегрузки (Congestion Control). * **Недостатки**: * Высокие накладные расходы (заголовок мин. 20 байт, процедура Handshake). * Задержки передачи (из-за подтверждений и таймаутов). * Не поддерживает Broadcast/Multicast.

UDP (User Datagram Protocol)

Протокол пользовательских датаграмм. Не ориентирован на соединение. * **Достоинства**: * Минимальные накладные расходы (заголовок 8 байт). * Высокая скорость (нет задержек на установление соединения и подтверждения). * Поддержка Broadcast и Multicast. * **Недостатки**: * Ненадежность (пакеты могут теряться, дублироваться). * Нет гарантии порядка (пакеты могут приходить вразнобой). * Нет контроля перегрузки сети.

Вопрос 50: Мультиплексирование. Использование fork для обработки клиентских соединений

Мультиплексирование

Технология, позволяющая одному процессу одновременно обрабатывать несколько потоков ввода-вывода (соединений). Вместо блокирования на чтении одного сокета, процесс опрашивает множество сокетов и работает с теми, где есть данные.

Модель с использованием fork()

Классическая модель “Один процесс — одно соединение”. **Алгоритм**: 1. Сервер слушает порт (`listen`). 2. При входящем соединении (`accept`) сервер делает системный вызов `fork()`. 3. **Родительский процесс**: Закрывает дескриптор соединения, продолжает ждать новых клиентов. 4. **Дочерний процесс**: Закрывает слушающий сокет, обрабатывает запрос клиента, затем завершается.

Плюсы: Изоляция клиентов (ошибка в одном не роняет сервер), простота кода. **Минусы**: Ресурсоемкость (на каждого клиента создается копия процесса), медленное переключение контекста, ограничение по числу процессов в ОС.

Вопрос 51: Мультиплексирование. Системные вызовы select, poll, epoll

select

Старейший системный вызов. Использует битовые маски (`fd_set`) для отслеживания дескрипторов. * **Ограничения**: Максимальное число дескрипторов ограничено константой `FD_SETSIZE` (обычно 1024). * **Сложность**: $O(N)$ — при каждом вызове нужно перебирать все биты и копировать маски из user-space в kernel-space и обратно.

poll

Использует массив структур `pollfd`. * **Отличия от select**: Нет жесткого лимита в 1024 дескриптора. * **Сложность**: $O(N)$ — при вызове ядро проходит по всему списку, чтобы проверить события. Эффективность падает линейно с ростом числа соединений.

epoll (Linux)

Событийно-ориентированный механизм. * **Принцип**: Дескрипторы добавляются в ядро один раз (`epoll_ctl`). Вызов `epoll_wait` возвращает список *только тех* сокетов, где произошли события. * **Сложность**: $O(1)$ относительно общего числа соединений. Самый эффективный метод для HighLoad серверов (Nginx, Node.js). * **Режимы**: * **Level Triggered (LT)**: Сообщает о событии, пока данные есть в буфере. * **Edge Triggered (ET)**: Сообщает о событии только один раз при поступлении данных.

Вопрос 52: Алгоритм инициализации сокета на сервере. Алгоритм инициализации сокета на клиенте

Алгоритм на сервере (TCP)

1. `socket()`: Создание сокета (получение файлового дескриптора). Указывается домен (IPv4/IPv6) и тип (STREAM для TCP).
2. `bind()`: Привязка сокета к локальному IP-адресу и порту.
3. `listen()`: Перевод сокета в режим ожидания входящих соединений. Создается очередь запросов на подключение.

Алгоритм на клиенте (TCP)

1. `socket()`: Создание сокета.
2. (Опционально) `bind()`: Обычно не используется, ОС сама назначает свободный порт и IP интерфейса.

Вопрос 53: Алгоритм установления соединения клиента с сервером. Алгоритм принятия соединения сервером от клиента

Установление соединения клиентом

1. Клиент вызывает `connect(socket, address)`.
2. Отправляется **SYN**-пакет серверу.
3. Клиент ждет **SYN-ACK** и отправляет **ACK** (TCP 3-way handshake).
4. Если успешно, функция возвращает 0, канал готов к `read`/`write`.

Принятие соединения сервером

1. Сервер вызывает `accept(listen_socket)`.
2. Функция блокирует выполнение (или возвращает ошибку в non-blocking), пока в очереди `listen` не появится полностью установленное соединение (после завершения handshake).
3. `accept` возвращает **новый** файловый дескриптор для связи с конкретным клиентом. Слушающий сокет остается свободным для приема новых.