Ответы на экзаменационные вопросы по Операционным системам

1. Общее представление об операционных системах

Местоположение ОС и её функции

Операционная система (ОС) — это совокупность пользовательских и системных программ, основной компонент которой является **ядро**. Ядро управляет ресурсами и осуществляет взаимодействие между оборудованием и приложениями.

Местоположение ОС:

- ОС занимает привилегированное положение между аппаратным обеспечением (hardware) и пользовательскими приложениями
- Ядро выполняется в привилегированном режиме (kernel mode), имея полный доступ к ресурсам
- Пользовательские программы работают в непривилегированном режиме (user mode)

Функции ОС:

1. **Управление ресурсами** — распоряжение физическими и абстрактными ресурсами, мультиплексирование во времени (доступ к CPU) и пространстве (доступ к памяти)
2. **Защита целостности данных** — изоляция процессов, защита пользователей и программ от несанкционированного доступа
3. **Предоставление абстракции** — увеличение уровня абстракции посредством предоставления единого интерфейса к оборудованию
4. **Управление памятью** — выделение, освобождение и защита адресных пространств процессов
5. **Управление процессами** — создание, планирование и завершение процессов
6. **Управление файлами и устройствами** — организация доступа к файловой системе и периферийным устройствам

Развитие операционных систем и причины появления

Эволюция ОС:

- **Батарейные системы** — программы запускались последовательно без ОС

- **Системы с разделением времени (Time-Sharing)** — одновременный доступ нескольких пользователей к одному компьютеру
- **Многопроцессные системы** — поддержка параллельного выполнения нескольких программ
- **Распределённые системы** — координация работы нескольких компьютеров в сети

Причины появления ОС:

- Необходимость эффективного использования дорогостоящих вычислительных ресурсов
- Потребность в стандартизации доступа к оборудованию
- Обеспечение многопользовательского и многопроцессного режима работы
- Упрощение разработки приложений за счет абстракции

2. Классификация операционных систем

Классификация по функциональным характеристикам

1. **По масштабу:**
   - Встроенные ОС (embedded systems) — минимальный функционал
   - Персональные ОС (Windows, macOS, Linux)
   - Серверные ОС — повышенная надёжность и производительность
   - Суперкомпьютерные ОС
2. **По числу пользователей:**
   - Однопользовательские — один пользователь одновременно
   - Многопользовательские — несколько пользователей одновременно
3. **По числу задач:**
   - Однозадачные (DOS) — одна программа одновременно
   - Многозадачные (современные) — несколько программ одновременно
4. **По числу процессов:**
   - Однопроцессные — один процессор
   - Многопроцессные — несколько процессоров
5. **По типу работы:**
   - Пакетные — обработка заданий в очереди
   - Интерактивные — взаимодействие с пользователем в реальном времени
   - Реального времени (RTOS) — гарантированные временные ограничения

Структурная классификация ОС

1. **По архитектуре ядра:**
   - Монолитное ядро
   - Микроядро

- Гибридное ядро
- Модульные ОС
- Слоистые ОС

3. Архитектуры ядер операционных систем

Монолитное ядро

Характеристики:

- Весь код ядра работает в одном адресном пространстве с максимальными привилегиями
- Все компоненты (файловая система, управление памятью, драйверы) тесно связаны
- Прямые вызовы функций между компонентами без промежуточных абстракций

Достоинства:

- Высокая производительность из-за минимальных накладных расходов на переходы между компонентами
- Простота реализации межкомпонентного взаимодействия
- Меньше переключений режимов

Недостатки:

- Низкая надёжность — ошибка в одном компоненте может привести к падению всей системы
- Сложность в разработке и отладке — всё переплетено
- Сложно расширять функционал
- Плохая модульность

Примеры: Linux, Unix, Windows NT (частично)

Микроядро

Характеристики:

- Ядро содержит только минимально необходимый функционал (управление процессами, память, IPC)
- Остальные компоненты (файловая система, драйверы) работают как обычные процессы в непривилегированном режиме
- Взаимодействие через механизм передачи сообщений (IPC)

Достоинства:

- Высокая надёжность — ошибка в одном сервисе не влияет на другие

- Хорошая модульность и расширяемость
- Возможность добавлять/удалять компоненты без перезагрузки
- Безопасность — каждый сервис в отдельном адресном пространстве

Недостатки:

- Низкая производительность из-за частых переключений режимов и передачи сообщений
- Высокие накладные расходы на IPC
- Сложность в реализации эффективного взаимодействия между сервисами

Примеры: Mach, QNX, MINIX

Гибридное ядро

Характеристики:

- Комбинация монолитного и микроядра
- Некритические функции работают как сервисы, критические остаются в ядре
- Стремление к компромиссу между производительностью и надёжностью

Достоинства:

- Лучшая производительность, чем чистое микроядро
- Лучшая надёжность, чем монолитное ядро
- Гибкость в выборе компонентов

Недостатки:

- Сложность в реализации
- Противоречивые требования к производительности и надёжности

Примеры: Windows NT, macOS (Darwin)

4. Режимы работы ядра

Привилегированный и непривилегированный режимы

Привилегированный режим (Kernel Mode):

- Процессор имеет полный доступ ко всем инструкциям и ресурсам
- Может выполнять специальные инструкции (lgdt, lidt, cli, sti и др.)
- Может обращаться к памяти и портам напрямую
- Может управлять прерываниями
- Используется только для ядра ОС

Непривилегированный режим (User Mode):

- Ограниченный набор инструкций
- Только инструкции общего назначения
- Нельзя обращаться к оборудованию напрямую
- Нельзя изменять регистры системного управления
- Все обращения к ресурсам выполняются через системные вызовы

Реализация на аппаратном уровне:

- На процессорах Intel x86 используется система колец защиты (protection rings)
- Обычно используется только Ring 0 (kernel mode) и Ring 3 (user mode)
- Текущий уровень привилегий хранится в регистре CPL (Current Privilege Level)

5. Понятие процесса

Определение и характеристики

Процесс — это динамический объект внутри вычислительной системы, представляющий собой экземпляр выполняемой программы с полным состоянием регистров процессора и выделенными ресурсами памяти и устройств.

Память процесса

Адресное пространство процесса состоит из секций:

1. **Stack (Стек)** — вверху адресного пространства
   - Хранит локальные переменные функций
   - Аргументы функций и адреса возврата
   - Растет вниз при добавлении данных
   - Максимальный размер обычно ограничен
2. **Heap (Куча)** — ниже стека
   - Динамически выделяемая память
   - malloc/free, new/delete
   - Растет вверх при выделении памяти
   - Более гибкая, но медленнее чем стек
3. **Data (Инициализированные данные)**
   - Глобальные и статические переменные
   - Инициализированы в программе
   - Известный размер на момент компиляции
4. **BSS (Неинициализированные данные)**
   - Глобальные и статические переменные без инициализации

- Инициализируются нулями при загрузке
- Занимает мало места в исполняемом файле

5. **Text (Код)**
    - Исполняемый код программы
    - Обычно read-only (защищено от записи)
    - Часто разделяется между процессами

Инициализация и завершение процесса

Создание процесса:

1. Порождение нового PCB с состоянием "New"
2. Присвоение уникального PID (Process ID)
3. Выделение ресурсов ОС
4. Загрузка кода программы в адресное пространство
5. Установка указателя инструкции на начало программы
6. Заполнение PCB
7. Переход в состояние "Ready"

Завершение процесса:

1. Изменение состояния на "Exit"
2. Освобождение ресурсов
3. Очистка PCB
4. Сохранение информации о причине завершения (код возврата)

Различие между процессами в Windows и Unix

Аспект	Unix	Windows
Создание	fork() — дублирование родителя	CreateProcess() — создание с нуля
Иерархия	Иерархическая структура	Независимые процессы
Совместное использование памяти	Шарится только код	Не шарится по умолчанию
PID родителя	Хранится PPID	Нет явного родителя
Наследование	Дочерний наследует ресурсы	Копирование ресурсов

Process Control Block (PCB)

PCB — это структура ядра, содержащая всю информацию об управлении процессом.

Содержание PCB:

Регистровый контекст:

- Состояние всех регистров процессора
- Указатель инструкции (PC/RIP)
- Регистр состояния (EFLAGS)

Системный контекст:

- Состояние процесса (New, Ready, Running, Blocked, Exit)
- PID, PPID, UID, GID
- Приоритет
- Информация о кванте времени
- Указатели на ресурсы

Контекст управления памятью:

- Указатель на таблицу страниц
- Границы сегментов памяти (text, data, bss, heap, stack)
- Размер памяти

Учетная информация:

- Пользователь, запустивший процесс
- Время создания, начала, конца выполнения
- Использованное процессорное время
- Причина последнего прерывания

Информация о файлах:

- Таблица дескрипторов открытых файлов
- Текущая рабочая директория

Информация о прерываниях:

- Обработчики сигналов
- Маска прерываний

6. Состояния процесса и таблица процессов

Состояния процесса

Основные состояния:

1. **New (Новый)**
   - Процесс только что создан
   - Еще не допущен к выполнению
2. **Ready (Готов)**
   - Процесс готов к выполнению
   - Ждет выделения процессорного времени
   - Находится в очереди готовых процессов
3. **Running (Исполняется)**
   - Процесс активно выполняется на процессоре
   - Имеет доступ к CPU
4. **Blocked/Waiting (Заблокирован)**
   - Процесс ждет события (I/O операция, срабатывание таймера)
   - Не может быть выполнен до наступления события
   - Находится в очереди ожидания события
5. **Exit (Завершен)**
   - Процесс завершил выполнение
   - Ожидает освобождения ресурсов

Переходы между состояниями:

- New → Ready: допуск к выполнению
- Ready → Running: выделение CPU (диспетчеризация)
- Running → Ready: истечение кванта времени (preemption)
- Running → Blocked: ожидание события
- Blocked → Ready: наступление события
- Running → Exit: нормальное завершение или ошибка

Таблица процессов

Таблица процессов — это структура ядра, содержащая PCB для всех процессов в системе.

Характеристики:

- Индексируется по PID
- Содержит указатели на PCB каждого процесса
- Ядро использует её для быстрого поиска процесса по PID

- Размер ограничен доступной памятью

Использование:

- Диспетчер задач использует таблицу для выбора следующего процесса
- При системных вызовах ядро ищет PCB текущего процесса в таблице
- При обработке прерываний используется информация из PCB

Моделирование режима многозадачности

Многозадачность реализуется через переключение контекста между процессами.

Процесс переключения контекста:

1. Сохранение состояния текущего процесса в его PCB
2. Выбор следующего процесса на основе алгоритма планирования
3. Загрузка состояния выбранного процесса из его PCB
4. Передача управления выбранному процессу

Квант времени (time slice):

- Определенный промежуток времени, в течение которого процесс может выполняться
- По истечении кванта происходит переключение на другой процесс
- Реализуется с помощью таймера и прерывания

7. Механизм прерываний и системные вызовы

Прерывания (Interrupts)

Прерывание — это событие, которое прерывает нормальное выполнение процесса.

Типы прерываний:

1. **Исключения (Exceptions)**
   - Аномальные ситуации, возникающие при выполнении инструкций
   - Деление на 0, обращение к защищённой памяти, инструкция привилегированного режима
   - Обработчик вызывается синхронно
2. **Аппаратные прерывания (Hardware Interrupts)**
   - Сигналы от устройств (клавиатура, диск, сеть)
   - Асинхронные относительно выполнения программы
   - Обрабатываются контроллером прерываний (PIC)
3. **Программные прерывания (Software Interrupts)**

- Вызваны инструкциями процессора (syscall, int на x86)
- Используются для системных вызовов
- Синхронные относительно программы

Обработка прерываний

Процесс обработки:

1. На процессоре используется **таблица прерываний (IDT — Interrupt Descriptor Table)**
   - Массив указателей на обработчики прерываний
   - 256 записей (вектор 0-255)
   - Адрес таблицы хранится в регистре IDTR
2. **При возникновении прерывания:**
   - Процессор сохраняет в стеке CS, RIP и EFLAGS
   - Загружает адрес обработчика из IDT по номеру прерывания
   - Переходит в режим ядра
   - Выполняет обработчик
3. **После обработки:**
   - Инструкция IRET восстанавливает CS, RIP и EFLAGS
   - Возврат к прерванному коду

Контексты исполнения программного кода

Контекст пользователя (User Context):

- Выполнение пользовательского кода
- Доступ к ресурсам через системные вызовы
- Непривилегированный режим

Контекст ядра (Kernel Context):

- Выполнение кода ядра при системном вызове
- Привилегированный режим
- Может обращаться ко всем ресурсам

Контекст обработки прерывания (Interrupt Context):

- Выполнение обработчика прерывания
- Привилегированный режим
- Прерывания могут быть запрещены

Доступ пользовательских программ к функциям ОС

Системные вызовы (System Calls):

- Единственный способ пользовательской программы обратиться к ресурсам
- На x86 используются инструкции `syscall` или `int 0x80`
- Ядро заполняет таблицу системных вызовов с указателями на функции ядра

Процесс системного вызова:

1. Пользовательская программа вызывает функцию из стандартной библиотеки
2. Библиотека подготавливает аргументы (обычно в регистрах)
3. Выполняется инструкция `syscall`
4. Происходит переключение в режим ядра
5. Ядро обрабатывает вызов
6. Результат возвращается в пользовательскую программу
7. Пользовательская программа продолжает работу

8. Кооперация процессов

Определение и причины

Кооперация процессов — это взаимодействие между двумя или более процессами для совместного решения задачи.

Причины кооперации:

- Повышение скорости решения задач (параллельное выполнение)
- Совместное использование данных между процессами
- Модульная конструкция системы (разделение функциональности)
- Удобство работы пользователя (shell piping, фильтры)

Способы межпроцессной коммуникации (IPC)

1. Каналы (Pipes)

- Однопроцессное, однонаправленное взаимодействие
- Буферизованный поток данных
- Синхронизация через блокировку при пустом/полном буфере

2. Сокеты (Sockets)

- Сетевое взаимодействие
- Поддержка TCP/UDP протоколов
- Может быть двунаправленным

3. Разделяемая память (Shared Memory)

- Несколько процессов обращаются к одной области памяти
- Быстрое взаимодействие
- Требует синхронизации доступа

4. Передача сообщений (Message Passing)

- Структурированный обмен данными
- Может быть синхронным или асинхронным
- Более безопасно, чем разделяемая память

5. Сигналы (Signals)

- Простое асинхронное оповещение
- Ограниченный по информативности механизм
- Не переносит большие объемы данных

6. Очереди сообщений (Message Queues)

- Асинхронный обмен структурированными сообщениями
- Декаплинг между отправителем и получателем

9. Каналы (Pipes)

Определение и характеристики

Pipe (конвейер/канал) — это один из самых простых механизмов взаимодействия между процессами в Unix.

Характеристики:

- **Однопроцессный**: обычно используется только между родительским и дочерним процессом
- **Однонаправленный (simplex)**: данные текут в одном направлении
- **Потоковая модель**: нет структуры, обменивается просто байтами
- **Буферизация**: небольшой буфер в ядре (обычно 4096 или 65536 байт)
- **Синхронизация**: процесс блокируется если пытается читать из пустого канала или писать в полный
- **Косвенная адресация**: по файловому дескриптору

Типы каналов

1. Неименованные каналы (Unnamed Pipes)

- Создаются системным вызовом `pipe()`

- Возвращают два файловых дескриптора (чтение и запись)
- Существуют только в памяти
- Доступны только процессам в одной иерархии

2. Именованные каналы (Named Pipes/FIFO)

- Создаются системным вызовом `mkfifo()`
- Имеют имя в файловой системе
- Персистентны (существуют до удаления)
- Могут использоваться процессами без кровного родства
- На Windows называются Named Pipes, на Unix — FIFO

Различия между Unix и Windows

Unix:

- Simplex каналы (однонаправленные)
- Обычно используются между родителем и потомком
- Имеют ограниченный буфер

Windows:

- Каналы могут быть full-duplex (двунаправленные)
- Могут использоваться между любыми процессами
- Лучшая поддержка сетевых каналов

Организация процессов в группы

Группы процессов (Process Groups):

- Несколько процессов организованы в группу
- Все процессы в группе получают одни и те же сигналы
- Используется shell для организации pipelines

Сессии (Sessions):

- Группа процессов объединяется в сессию
- Сессия имеет лидера (процесс сессии)
- Используется для управления терминалом и работой заданий

10. Классификация систем по Флинну

SISD (Single Instruction, Single Data)

- Одна инструкция обрабатывает один элемент данных
- Одноядерные системы
- Параллельности нет
- Пример: старые процессоры, однопроцессные компьютеры

SIMD (Single Instruction, Multiple Data)

- Одна инструкция обрабатывает несколько элементов данных одновременно
- Векторные процессоры, графические процессоры (GPU)
- Расширения: MMX, SSE, AVX (x86), Neon (ARM)
- Пример: обработка изображений, научные вычисления

MISD (Multiple Instruction, Single Data)

- Несколько инструкций обрабатывают один элемент данных
- Отказоустойчивые системы
- Редко встречается, экзотика
- Несколько потоков выполняют разные инструкции над одним потоком данных

MIMD (Multiple Instruction, Multiple Data)

- Несколько инструкций обрабатывают несколько элементов данных
- Многоядерные процессоры, кластеры, суперкомпьютеры
- Полный, честный параллелизм
- Наиболее универсальные и распространённые архитектуры

Локальные и распределённые системы

Локальные системы (Shared Memory):

- Процессы работают на одной машине
- Совместное использование оперативной памяти
- Более быстрое взаимодействие
- Меньше накладных расходов

Распределённые системы (Distributed Systems):

- Процессы работают на разных машинах

- Взаимодействие через сеть
- Сложнее синхронизация
- Выше надёжность при отказе отдельных узлов

11. Понятие потока (Thread)

Определение и причины создания

Поток (Thread) — это единица исполнения программного кода в адресном пространстве процесса. Это легковесный процесс, работающий в общем адресном пространстве с другими потоками.

Причины создания потоков:

- **Производительность**: создание потока быстрее, чем создание процесса
- **Синхронизация данных**: потоки в одном процессе легче обмениваются данными через общую память
- **Эффективность на многоядерных системах**: потоки могут работать параллельно на разных ядрах
- **Отзывчивость приложения**: в пользовательском интерфейсе один поток может обрабатывать события, другой выполнять работу

Свойства потоков

Общие с процессом:

- Состояние процесса (Ready, Running, Blocked, Exit)
- Указатель инструкции (Program Counter)
- Содержание регистров
- Данные для планирования использования CPU

Собственные:

- Каждый поток имеет собственный стек
- Собственный набор регистров
- Единицей планирования в ядре ОС является именно поток, а не процесс!

Совместные между потоками в процессе:

- Адресное пространство процесса (text, data, bss, heap)
- Таблица дескрипторов открытых файлов
- Окружение (environment variables)
- Контекст сигналов

12. Реализация сервера для обработки запросов

Однопоточный процесс vs множество потоков

Однопоточный процесс:

1. Ждем подключения клиента (блокирующий accept)
2. Обрабатываем одного клиента
3. Закрываем соединение
4. Возвращаемся к ожиданию подключения

Недостатки:

- Может обслуживать только одного клиента одновременно
- Другие клиенты ждут в очереди

Множество потоков:

1. Основной поток ждет подключения клиентов
2. При подключении клиента создается новый поток
3. Новый поток обрабатывает клиента
4. Основной поток продолжает ждать новых клиентов
5. Несколько клиентов могут обслуживаться параллельно

Достоинства многопоточного подхода:

- Параллельная обработка нескольких клиентов
- Лучше использование многоядерных систем
- Меньше времени ожидания для клиентов

13. Стратегии реализации потоков

Потоки уровня ядра (Kernel-Level Threads, KLT)

Характеристики:

- Полностью управляются ядром ОС
- Ядро имеет информацию о каждом потоке
- Каждый поток имеет собственную запись в таблице потоков ядра
- Переключение потоков выполняется ядром

Достоинства:

- Высокая степень параллелизма и устойчивость к ошибкам
- Один заблокированный поток не блокирует другие
- Истинный параллелизм на многоядерных системах

Недостатки:

- Высокие накладные расходы на переключение контекста (переход в режим ядра)
- Переключение потоков более затратно, чем переключение между потоками уровня пользователя

Потоки уровня пользователя (User-Level Threads, ULT)

Характеристики:

- Управляются библиотекой в пользовательском пространстве
- Ядро не знает о них (видит только процесс)
- Переключение потоков без вмешательства ядра
- Планирование выполняется пользовательской библиотекой

Достоинства:

- Высокая производительность переключения потоков
- Гибкость в реализации алгоритма планирования
- Минимальные накладные расходы

Недостатки:

- Блокировка одного потока может заблокировать весь процесс
- Невозможно истинный параллелизм (один процесс на одном ядре)
- Если один поток выполняет I/O операцию, процесс может остановиться

Гибридная модель (Hybrid/M:N Model)

Характеристики:

- Комбинация KLT и ULT
- M пользовательских потоков мультиплексируются на N ядерных потоков
- Каждый ядерный поток может содержать несколько пользовательских потоков

Достоинства:

- Баланс между производительностью и параллелизмом
- Гибкость и масштабируемость
- Один заблокированный поток не всегда блокирует всё

Недостатки:

- Сложность в реализации
- Сложность в отладке

14. Fork-and-Join модель параллельного программирования

Принцип:

- Основной поток создает (fork) новые потоки для выполнения параллельных задач
- Основной поток ждет (join) завершения всех созданных потоков
- После завершения всех потоков продолжается основной код

Жизненный цикл:

1. Основной поток (родитель)
2. Fork: создание N дочерних потоков
3. Параллельное выполнение всех потоков
4. Join: ожидание завершения всех потоков
5. Продолжение основного потока

Особенности:

- Нет ограничений на количество создаваемых потоков
- Простая модель для понимания
- Удобна для задач, разбиваемых на независимые подзадачи

Проблемы:

- Если создать потоков больше, чем логических процессоров, наблюдается **псевдопараллелизм**
- Частые операции создания и уничтожения потоков добавляют накладные расходы
- Не оптимально для задач с частым созданием новых потоков

15. Псевдопараллелизм

Определение: ситуация, когда создано больше потоков, чем логических процессоров в системе.

Проблемы:

- Ядру ОС приходится планировать и переключать больше потоков, чем ядер доступно

- Значительные накладные расходы на контекстные переключения
- Если количество потоков намного больше количества ядер, производительность падает
- Может быть даже медленнее, чем последовательное выполнение

Пример:

- Система с 4 ядрами
- Создано 100 потоков
- Каждое ядро должно быстро переключаться между потоками
- Большая часть времени тратится на переключение контекста вместо полезной работы

16. Ошибки в параллельном программировании

Гонка данных (Data Race)

Определение: две или более операции попытаются одновременно обратиться к одной области памяти, и хотя бы одна из них — запись.

Проблема:

- Результат зависит от порядка выполнения операций
- Порядок не определен и может меняться
- Программа работает неправильно и непредсказуемо

Пример:

```
volatile int x = 0;
// Поток 1:           // Поток 2:
mov eax, [x]          mov eax, [x]
inc eax                inc eax
mov [x], eax          mov [x], eax
```

Ожидаемо: x = 2, но на деле может быть x = 1 в зависимости от порядка выполнения.

Взаимоблокировка (Deadlock)

Определение: ситуация, когда два или более потока бесконечно ждут друг друга и не могут продолжить выполнение.

Условия возникновения:

1. Взаимное исключение (mutual exclusion)
2. Захват и удержание ресурсов (hold and wait)

3. Отсутствие предотивности (no preemption)
4. Циклическое ожидание (circular wait)

Способы избежания:

- Упорядочить захват ресурсов (всегда в одном порядке)
- Использовать таймауты при блокировке
- Применять алгоритмы избежания взаимоблокировок (банкир)

17. Метрики параллельных алгоритмов

Ускорение (Speedup)

Определение:

```
S = Ts / Tp
```

где:

- Ts — время последовательной реализации
- Tp — время параллельной реализации на p ядрах
- S — ускорение

Свойства:

- 1 ≤ S ≤ p (идеально S = p)
- Если S = 1, параллелизм не дает выигрыша
- Если S = p, достигнут идеальный параллелизм (линейное ускорение)
- Если S > p, есть какая-то ошибка (обычно неправильное измерение Ts)

Эффективность (Efficiency)

Определение:

```
E = S / p
```

где p — количество ядер.

Свойства:

- 0 < E ≤ 1
- E = 1 означает идеальное использование всех ядер
- E близко к 0 означает плохое использование параллелизма

18. Закон Мура (Moore's Law)

Формулировка: количество транзисторов на кристалле удваивается примерно каждые два года.

Следствия:

- Рост производительности процессоров прекратился из-за физических ограничений (тепловыделение, размер транзисторов)
- Переход к многоядерным системам для увеличения производительности
- Теперь нужно эффективнее использовать существующие ресурсы

19. Закон Амдала (Amdahl's Law)

Формулировка:

```
S = 1 / (a + (1-a)/p)
```

где:

- a — доля последовательного кода ($0 < a \leq 1$)
- p — количество ядер
- S — ускорение

Интерпретация:

- Даже малый процент последовательного кода ограничивает ускорение
- Если a = 0.1 (10% последовательного кода) и p = 100 ядер, то $S \leq 10.9$
- Улучшение параллельной части дает уменьшающиеся выигрыши

Пример:

- Программа с 10% последовательного кода
- На 2 ядрах: S = 1/(0.1 + 0.9/2) = 1.82
- На 10 ядрах: S = 1/(0.1 + 0.9/10) = 5.26
- На 100 ядрах: S = 1/(0.1 + 0.9/100) = 9.17

20. Критическая область (Critical Section)

Определение: участок кода, где один поток получает монопольный доступ к общему ресурсу.

Структура:

1. Вход в критическую область (захват блокировки)
2. Исполнение критического кода
3. Выход из критической области (освобождение блокировки)

Требования:

- Взаимное исключение (только один поток одновременно)
- Прогресс (если никто не в критической области, любой потом должен войти)
- Справедливость (все потоки получат доступ)

Проблемы:

- Использование критических областей увеличивает последовательную часть кода
- По закону Амдала это уменьшает ускорение
- Следует минимизировать размер и количество критических областей

21. Взаимоисключение с активным ожиданием

Активное ожидание (Busy Waiting): процесс постоянно проверяет условие в цикле, вместо того чтобы заснуть.

Простое решение с флагом

```
struct Mutex {
    bool flag[2];
};

void lock(Mutex *mutex) {
    int i = gettid();
    int j = 1 - i;
    mutex->flag[i] = true;
    while (mutex->flag[j]);  // Активное ожидание
}

void unlock(Mutex *mutex) {
    int i = gettid();
    mutex->flag[i] = false;
}
```

Проблема: This doesn't work! (взаимоблокировка возможна)

Решение с переменной victim

```
struct Mutex {
    int victim;
};

void lock(Mutex *mutex) {
    int i = gettid();
    mutex->victim = i;
    while (mutex->victim == i);
}

void unlock(Mutex *mutex) {
    // Ничего не делаем
}
```

Проблема: два потока могут одновременно быть в критической области

Алгоритм Петерсона (Peterson's Algorithm)

```
struct Mutex {
    bool flag[2];
    int victim;
};

void lock(Mutex *mutex) {
    int i = gettid();
    int j = 1 - i;
    mutex->flag[i] = true;
    mutex->victim = i;
    while (mutex->flag[j] && mutex->victim == i);
}

void unlock(Mutex *mutex) {
    int i = gettid();
    mutex->flag[i] = false;
}
```

Достоинства:

- Гарантирует взаимное исключение
- Нет взаимоблокировок
- Справедлив

Недостатки:

- Активное ожидание (бесполезно тратит CPU)
- Работает только для 2 потоков
- Для N потоков алгоритм Петерсона становится сложнее

22. Семафор, мьютекс и условная переменная

Семафор (Semaphore)

Определение: примитив синхронизации с счетчиком для управления доступом к ограниченному количеству ресурсов.

Операции:

- `wait()` — декрементирует счетчик, если > 0, иначе блокируется
- `post()` — инкрементирует счетчик и пробуждает один ждущий поток

Использование:

```
semaphore sem = 3;  // 3 ресурса
wait(&sem);         // Получить ресурс
// Использовать ресурс
post(&sem);         // Освободить ресурс
```

Типы:

- Двоичный семафор (счетчик 0 или 1) — как мьютекс
- Счетный семафор — для ограничения доступа к N ресурсам

Мьютекс (Mutex)

Определение: простейший примитив синхронизации для предоставления монопольного доступа к одному ресурсу.

Состояния:

- Разблокирован (0) — никто не владеет
- Заблокирован (1) — владеет один поток

Операции:

- `lock()` — захватить мьютекс (блокировка, если занят)
- `unlock()` — освободить мьютекс (разбудить один потом)

Отличие от семафора:

- Мьютекс может владеть только тот поток, который его захватил
- Семафор может быть освобожден любым потоком
- Мьютекс ориентирован на защиту ресурса одним потоком
- Семафор управляет доступом к пулу ресурсов

Условная переменная (Condition Variable)

Определение: примитив для координации потоков, позволяющий одним потокам ждать выполнения условия, установленного другими потоками.

Операции:

- `wait(mutex)` — освобождает мьютекс и ждет сигнала
- `broadcast()` — разбудить все ждущие потоки
- `signal()` — разбудить один ждущий поток

Использование:

```
while (!condition) {
    pthread_cond_wait(&cond, &mutex);
}
// Условие выполнено, можем продолжить

// В другом потоке:
condition = true;
pthread_cond_broadcast(&cond);
```

Отличие условной переменной от семафора:

- Условная переменная не хранит сигналы (если broadcast до wait, wait заблокируется)
- Семафор запоминает post операции
- Условная переменная требует мьютекса для защиты состояния
- Семафор самостоятельный

23. Атомарные переменные и CAS инструкция

Атомарные операции

Определение: низкоуровневые операции, выполняемые за один такт процессора без прерывания другими потоками.

Используемые инструкции:

- Чтение (Load)
- Запись (Store)
- Инкремент/Декремент (Inc/Dec)
- Сравнение-и-Обмен (CAS — Compare-and-Swap)
- Обмен (Exchange)

Свойства:

- Зависят от архитектуры процессора
- На x86 используются префиксы `lock` для гарантии атомарности
- Процессор останавливает все остальные ядра на время операции
- Дорогие в выполнении

CAS инструкция (Compare-And-Swap)

Операция:

```
bool CAS(int *address, int expected, int new_value) {
    if (*address == expected) {
        *address = new_value;
        return true;
    }
    return false;
}
```

На ассемблере (x86):

```
lock cmpxchg [rdi], rbx
```

Использование:

```
atomic_int counter = 0;
int old_value = 0;
while (!CAS(&counter, old_value, old_value + 1)) {
    old_value = counter;  // Получить новое значение и повторить
}
```

Спин-мьютекс (Spin Mutex)

Определение: мьютекс с активным ожиданием на основе атомарных операций.

Реализация:

```
struct SpinMutex {
    volatile bool flag;
};

void lock_spin_mutex(SpinMutex *spin) {
    while (!CAS(&spin->flag, false, true)) {
        // Активное ожидание (spining)
        pause();  // Инструкция PAUSE для снижения нагрузки
    }
}

void unlock_spin_mutex(SpinMutex *spin) {
    spin->flag = false;
}
```

Использование:

- Если критическая область очень мала и не содержит системных вызовов
- Спин-мьютекс эффективнее, чем обычный (избегает переключения контекста)
- На многопроцессорных системах спин-мьютекс может быть быстрее

24. Передача сообщений и барьеры

Передача сообщений (Message Passing)

Определение: механизм синхронизации, где потоки обмениваются структурированными сообщениями.

Характеристики:

- Могут быть синхронные или асинхронные
- Обмен структурированными данными (не просто числами)
- Более безопасно, чем разделяемая память (не требует явной синхронизации)

Использование:

- Параллельные алгоритмы, где потоки выполняют независимые задачи
- Распределённые системы

Барьер (Barrier)

Определение: примитив синхронизации, который синхронизирует N потоков в определённой точке кода.

Принцип:

- N потоков должны дошли до барьера
- Ни один поток не может пройти дальше, пока все N потоков не достигли барьера
- Когда все N потоков достигли барьера, они все разблокируются одновременно

Использование:

```
pthread_barrier_init(&barrier, NULL, N);  // Инициализация для N потоков
// ... работа ...
pthread_barrier_wait(&barrier);  // Ждем остальных N-1 потоков
// Все потоки продолжают отсюда одновременно
```

25. Взаимоблокировка

Условия возникновения

Для возникновения взаимоблокировки необходимы все 4 условия одновременно:

1. **Взаимное исключение** — ресурс может использовать только один процесс одновременно
2. **Захват и удержание** — процесс, удерживающий ресурс, может захватить дополнительные ресурсы
3. **Отсутствие предотивности** — ресурс не может быть силой забран у процесса
4. **Циклическое ожидание** — существует цепочка процессов, где каждый ждет ресурса у следующего

Способы борьбы со взаимоблокировками

1. Предотвращение (Prevention):

- Нарушить одно из четырех условий
- Например, упорядочить захват ресурсов (нарушает циклическое ожидание)
- Или разрешить предотивность ресурсов

2. Избежание (Avoidance):

- Алгоритм банкира — анализирует состояние и предпринимает попытку захвата только, если она безопасна

- Требует заранее знать максимальное требование каждого потока

3. Обнаружение и восстановление (Detection and Recovery):

- Периодически проверять наличие циклов в графе ожидания ресурсов
- При обнаружении взаимоблокировки убить один из процессов или откатить транзакцию

4. Игнорирование:

- Самый простой подход
- Надеяться, что взаимоблокировка не произойдет
- Или обработать её вручную (таймауты)

26. Управление памятью

Адресные пространства

Виртуальное адресное пространство:

- Абстрактное адресное пространство, которое видит процесс
- Линейное, от 0 до максимального адреса
- Каждый процесс имеет собственное адресное пространство
- Защита: один процесс не может обращаться к памяти другого

Физическое адресное пространство:

- Реальные адреса в оперативной памяти
- Разделяется между всеми процессами
- Управляется ядром ОС

MMU (Memory Management Unit):

- Аппаратный блок процессора
- Переводит виртуальные адреса в физические
- Выполняет проверки доступа

Свопинг (Swapping)

Определение: техника расширения доступной памяти путём выгрузки части памяти на диск.

Механизм:

- Если физической памяти недостаточно, ОС выгружает (swaps out) неиспользуемые страницы памяти на диск (swap file)

- Когда программе нужна выгруженная страница, ОС загружает (swaps in) её обратно в память
- Происходит прозрачно для приложения

Проблемы:

- Диск намного медленнее памяти (в 1000+ раз)
- Частый свопинг существенно замедляет систему
- Современные ОС стараются минимизировать свопинг

Общие принципы управления памятью

1. **Выделение памяти**
   - На момент создания процесса или при запросе
   - Динамическое выделение (malloc) по требованию
2. **Защита памяти**
   - Один процесс не может обращаться к памяти другого
   - Осуществляется через виртуальную память и права доступа
3. **Освобождение памяти**
   - При завершении процесса
   - При явном запросе (free)
   - Автоматически в языках с garbage collection

Распределитель памяти ядра (Kernel Memory Allocator)

Функции:

- Выделяет память для структур ядра (PCB, таблицы страниц и т.д.)
- Должен быть быстрым и эффективным
- Должен минимизировать фрагментацию

Типы аллокаторов:

- Buddy system — делит память на блоки степеней 2
- Slab allocator — кэширует часто используемые объекты
- First fit, best fit, worst fit — стратегии поиска подходящего блока

Оценка аллокаторов

Критерии:

- Скорость выделения/освобождения
- Фрагментация памяти
- Эффективность использования памяти

- Поддержка многопоточности

Способы отслеживания свободной памяти

1. **Битовая карта (Bitmap)**
   - Каждый бит представляет один блок памяти
   - 0 — свободно, 1 — занято
   - Простая, но требует много памяти для больших адресных пространств
2. **Связанный список (Linked List)**
   - Список свободных (или занятых) блоков
   - Каждый элемент содержит адрес и размер
   - Гибкая, но медленнее для поиска
3. **Дерево (Tree)**
   - Организация блоков в иерархическую структуру
   - Быстрее поиск

27. Memory Mapping

Определение: техника ядра ОС, позволяющая отобразить содержимое файла в виртуальное адресное пространство процесса.

Преимущества:

- Файл становится доступен как обычная память
- Нет необходимости в read/write системных вызовах
- Ленивая загрузка — содержимое файла подгружается по мере обращения
- Несколько процессов могут отобразить один файл и разделить память
- Автоматическая синхронизация изменений с файлом

Использование:

```
void *addr = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);
// Теперь можно работать с памятью как с обычным массивом
*(int*)addr = 42;
// ...
munmap(addr, size);
```

28. Загрузчик программ (Program Loader)

Секции памяти программ

ELF (Executable and Linkable Format) структура:

1. **Text (.text)**
   - Исполняемый код
   - Read-only, часто shared между процессами
   - Загружается с флагами PROT_READ | PROT_EXEC
2. **Initialized Data (.data)**
   - Инициализированные глобальные и статические переменные
   - Read-write, private (CoW)
   - Загружается с флагами PROT_READ | PROT_WRITE
3. **Uninitialized Data (.bss)**
   - Неинициализированные глобальные и статические переменные
   - Инициализируются нулями при загрузке
   - Занимает место в памяти, но не в файле
4. **Read-only Data (.rodata)**
   - Строковые константы, const переменные
   - Read-only, shared
   - Загружается с флагом PROT_READ

Алгоритм загрузки программы

Шаги при вызове exec():

1. **Парсинг ELF заголовка**
   - Проверка магического числа (0x7F 'E' 'L' 'F')
   - Проверка архитектуры, типа файла (executable/shared object)
2. **Очистка адресного пространства**
   - Освобождение старого кода и данных процесса
   - Сохранение PID и других метаданных
3. **Отображение секций в память**
   - Использование mmap() для отображения каждой секции
   - .text: PROT_READ | PROT_EXEC, MAP_SHARED
   - .data/.bss: PROT_READ | PROT_WRITE, MAP_PRIVATE
   - Адреса выбираются с учётом ASLR (Address Space Layout Randomization)
4. **Загрузка динамического загрузчика**
   - Если программа динамически скомпонована
   - Загружается ld-linux.so

- Загрузчик затем загружает зависимые библиотеки

5. **Подготовка стека**
   - Выделение стека в верхней части адресного пространства
   - Помещение на стек:
     - Аргументы командной строки (argv)
     - Переменные окружения (envp)
     - Auxiliary vector (auxv) — адрес загрузчика, ASLR данные
6. **Настройка VDSO**
   - Virtual Dynamic Shared Object для быстрых syscall'ов
7. **Установка точки входа**
   - Установка RIP на e_entry из ELF заголовка (обычно _start)
8. **Переход в user mode**
   - Возврат в непривилегированный режим
   - Передача управления _start()

29. Статическая и динамическая линковка

Статическая линковка

Процесс:

1. Compile time: компилирование отдельных объектных файлов
2. Link time: компоновщик (linker) разрешает все символы и создаёт один исполняемый файл
3. Runtime: образ исполняемого файла загружается с фиксированными адресами

Достоинства:

- Независимость — все зависимости уже разрешены
- Нет необходимости в дополнительных библиотеках на целевой системе
- Быстрая загрузка — нет разрешения символов

Недостатки:

- Больший размер исполняемого файла (код библиотек встроен)
- Сложно обновлять библиотеки — нужно перекомпилировать приложение
- Дублирование кода в памяти, если несколько приложений используют одну библиотеку

Динамическая линковка

Процесс:

1. Link time: компоновщик оставляет неразрешённые символы, указывая требуемые библиотеки

2. Runtime: динамический загрузчик загружает библиотеки и разрешает символы

Достоинства:

- Меньший размер исполняемого файла
- Легко обновлять библиотеки без перекомпиляции приложения
- Экономия памяти — одна библиотека в памяти для нескольких приложений
- Гибкость

Недостатки:

- Усложненная загрузка программы (разрешение символов)
- Зависимость от наличия библиотек на целевой системе
- Проблемы с совместимостью версий (DLL hell, library version conflicts)

Динамическая загрузка библиотек

Загрузка во время выполнения:

```
#include <dlfcn.h>

void *library = dlopen("./libc.so", RTLD_LAZY);
typedef int (*func_t)(int);
func_t func = (func_t)dlsym(library, "function_name");
int result = func(42);
dlclose(library);
```

Флаги dlopen:

- RTLD_LAZY — разрешить символы при необходимости
- RTLD_NOW — разрешить все символы сразу
- RTLD_GLOBAL — сделать символы доступными для других библиотек
- RTLD_LOCAL — символы доступны только через dlsym

30. Таблицы PLT и GOT

PLT (Procedure Linkage Table)

Определение: таблица в исполняемом файле, содержащая заглушки для вызовов функций из динамических библиотек.

Структура:

- Каждая запись соответствует одному вызову внешней функции

- Создается компоновщиком при компилировании с -fPIC

Механизм работы:

1. Программа вызывает функцию `printf@plt`
2. PLT запись указывает на GOT запись (ленивое связывание, lazy binding)
3. Первый вызов:
    - PLT передает управление в динамический загрузчик
    - Загрузчик разрешает адрес функции
    - Записывает адрес в GOT
4. Последующие вызовы:
    - PLT напрямую использует адрес из GOT
    - Больше не нужен загрузчик

GOT (Global Offset Table)

Определение: таблица, содержащая абсолютные адреса функций и глобальных переменных.

Использование:

- Библиотеки должны работать в любом адресном пространстве
- Используются относительные адреса к GOT
- Адреса функций хранятся в GOT
- При загрузке библиотеки динамический загрузчик заполняет GOT

Двуслойная система:

1. Код обращается к PLT (которая находится рядом с кодом)
2. PLT обращается к GOT (которая содержит точные адреса)

31. Позиционно-независимый код (PIC)

Определение: код, который может быть загружен в любое место в памяти без изменений.

Как это достигается:

- Все обращения к функциям и данным происходят через RIP-относительную адресацию
- На x86-64: инструкции используют register RIP (текущий адрес инструкции)
- Пример: `lea rax, [rel func]` — загрузить адрес func относительно текущего RIP

Компиляция:

```
gcc -o lib.so -shared -fPIC lib.c
```

Флаг -fPIC:

- Создает PLT и GOT сегменты
- Патчит адреса для использования RIP-относительной адресации
- Без -fPIC библиотека может работать неправильно

Преимущества:

- Несколько процессов могут использовать одну копию библиотеки в памяти
- Повышение безопасности (ASLR)

32. Общая память процессов

Определение: область памяти, доступная нескольким процессам одновременно.

Механизмы:

- System V IPC (shmget, shmat, shmdt)
- POSIX Shared Memory (shm_open, mmap)
- Memory mapping (mmap с MAP_SHARED флагом)

Синхронизация:

- Требуется механизм синхронизации (мьютекс, семафор)
- Обычно используется POSIX IPC элементы

Использование:

```
// Создание разделяемой памяти
int shm_id = shmget(key, size, IPC_CREAT | 0666);
void *addr = shmat(shm_id, NULL, 0);
// Запись в разделяемую память
*(int*)addr = 42;
// Открепление
shmdt(addr);
```

33. Страничная организация памяти и виртуальная память

Общие положения

Виртуальная память:

- Абстракция, позволяющая процессам использовать больше памяти, чем физически доступно
- Каждый процесс имеет виртуальное адресное пространство от 0 до максимума
- Ядро отображает виртуальные адреса на физические (или на диск)

Страничная организация:

- Виртуальная память делится на блоки фиксированного размера — страницы (обычно 4KB)
- Физическая память делится на блоки того же размера — фреймы
- Таблица страниц содержит отображение виртуальных страниц на физические фреймы

Таблица страниц

Структура на x86-64 (4-уровневая):

1. PML4 (Page Map Level 4) — первый уровень
2. PDPT (Page Directory Pointer Table) — второй уровень
3. PDT (Page Directory Table) — третий уровень
4. PT (Page Table) — четвёртый уровень

Трансляция адреса:

1. Первые 12 бит виртуального адреса — смещение в странице (4KB)
2. Следующие 9 бит — индекс в PT
3. Следующие 9 бит — индекс в PDT
4. Следующие 9 бит — индекс в PDPT
5. Последние 9 бит — индекс в PML4

Процесс трансляции:

1. Прочитать CR3 регистр (адрес PML4)
2. Использовать первые 9 бит для индексирования PML4
3. Получить адрес PDPT
4. Повторить для остальных таблиц
5. Получить физический адрес фрейма
6. Добавить смещение внутри страницы

TLB (Translation Lookaside Buffer)

Определение: кэш таблицы страниц внутри процессора.

Характеристики:

- Кэширует недавние трансляции виртуального адреса → физический адрес

- Очень быстрый доступ (1-2 цикла)
- Небольшой размер (64-1024 записей)
- При переключении контекста часто очищается (TLB flush)

Эффективность:

- При хорошей локальности обращений TLB попадание происходит часто
- При плохой локальности TLB промахи частые, и требуется обращение к памяти

Поддержка большого объема памяти

Методы:

1. **Увеличение размера страницы**
   - Большие страницы (2MB, 1GB на x86-64)
   - Меньше записей в таблице страниц
   - Но увеличивается внутренняя фрагментация
2. **Увеличение размера таблицы страниц**
   - Добавление уровней индирекции (что делает x86-64)
   - Требует больше памяти для самих таблиц
3. **Инвертированные таблицы страниц**
   - Индексируются по физическому адресу вместо виртуального
   - Требуют дополнительной хэш-таблицы для поиска

34. Оптимизации при работе со страничной памятью и сегментирование

Working Set

Определение: набор страниц, активно используемых процессом в текущий момент.

Оптимизация:

- Если working set полностью находится в памяти, процесс работает быстро
- Если working set больше доступной памяти, много page fault'ов
- ОС старается держать весь working set в памяти

Сегментирование (Segmentation)

Определение: разделение виртуального адресного пространства на логические сегменты с разными правами доступа.

Типы сегментов:

- Code segment — только чтение и выполнение
- Data segment — чтение и запись
- Stack segment — особые правила роста

Преимущества:

- Защита — разные права для разных сегментов
- Экономия памяти — размер сегмента может быть динамическим

Недостатки:

- Усложнение управления памятью
- Современные системы используют в основном пейджинг, сегментирование отходит на второй план

35. Файловые системы

Общие положения

Цели файловой системы:

- Предоставить единый интерфейс для работы с блочными устройствами
- Организовать иерархию файлов и директорий
- Управлять пространством на диске
- Обеспечить защиту и права доступа

Понятие файла

Файл — это именованная последовательность байтов на диске.

Типы файлов:

- Обычный файл — содержит пользовательские данные
- Директория — содержит список файлов
- Символическая ссылка — указатель на другой файл
- Устройство — доступ к аппаратным устройствам
- Сокет — для сетевого взаимодействия
- FIFO — именованный канал

Структура файла

На уровне файловой системы:

- Данные файла организованы в блоки

- i-узел содержит метаданные файла
- Размер файла хранится в i-узле

Типы и характеристики файлов

Характеристика	Значение
Имя	текстовое имя файла
Тип	тип (файл, директория, ...)
Размер	в байтах
Расположение	адреса блоков на диске
Владелец	UID
Группа	GID
Права доступа	rwxrwxrwx + флаги
Время доступа	atime
Время изменения	mtime/ctime

Понятие директории (каталога)

Директория — это специальный файл, содержащий таблицу соответствия между именами файлов и их i-узлами.

Структура директории:

```
имя_файла1 → номер_i-узла1
имя_файла2 → номер_i-узла2
...
```

Операции над файлами и каталогами

Над файлами:

1. `open()` — открыть файл
2. `close()` — закрыть файл
3. `read()` — прочитать данные
4. `write()` — записать данные
5. `lseek()` — переместить указатель
6. `unlink()` — удалить файл

7. `rename()` — переименовать
8. `chmod()`, `chown()` — изменить права и владельца
9. `stat()` — получить атрибуты

Над каталогами:

1. `mkdir()` — создать директорию
2. `rmdir()` — удалить (пустую) директорию
3. `opendir()`, `closedir()` — открыть/закрыть директорию
4. `readdir()` — прочитать следующий элемент
5. `chdir()` — изменить текущую директорию
6. `link()` — создать жёсткую ссылку
7. `symlink()` — создать символическую ссылку

36. Реализация файлов в файловой системе

Непрерывное размещение файлов

Принцип: все блоки файла находятся подряд на диске.

Достоинства:

- Очень быстрое чтение последовательного доступа
- Минимизация head movements на диске

Недостатки:

- Фрагментация при удалении файлов
- Сложно расширять файл (может не быть свободного места рядом)
- Требуется дефрагментация

I-узлы

i-node (index node) — структура, содержащая все метаданные файла, кроме имени.

Содержание i-узла:

- Тип файла (обычный, директория, ссылка...)
- Владелец (UID, GID)
- Права доступа (rwxrwxrwx + флаги)
- Размер файла в байтах
- Количество жёстких ссылок (link count)
- Время доступа, изменения, создания (atime, mtime, ctime)

- Указатели на блоки данных
- Флаги (immutable, append-only, noatime и др.)

Указатели на блоки в i-узле:

- Первые N указателей — прямые (direct) указатели на блоки
- Следующие указатели — косвенные (indirect):
  - Одноуровневый косвенный указатель
  - Двухуровневый косвенный указатель
  - Трёхуровневый косвенный указатель

Пример структуры i-узла (ext2):

```
struct ext2_inode {
    uint16_t i_mode;        // Тип + права
    uint16_t i_uid;         // Владелец
    uint32_t i_size;        // Размер файла
    uint32_t i_blocks;      // Количество блоков
    uint32_t i_block[15];   // Указатели на блоки
    // i_block[0-11] – прямые
    // i_block[12] – одноуровневый косвенный
    // i_block[13] – двухуровневый косвенный
    // i_block[14] – трёхуровневый косвенный
};
```

Совместно используемые файлы

Жёсткие ссылки (Hard Links):

- Несколько имён указывают на один и тот же i-узел
- Все имена равноправны
- При удалении одного имени файл остается (пока не удалены все имена)
- Link count в i-узле отслеживает количество имён

Символические ссылки (Symbolic Links/Symlinks):

- Специальный файл, содержащий путь к другому файлу
- Если исходный файл переименован, ссылка становится недействительной
- Может указывать на файлы в других файловых системах
- Может указывать на несуществующие файлы

37. Жёсткие и символические ссылки

(см. выше в разделе про i-узлы)

38. Суперблок как описатель файловой системы

Суперблок — структура, содержащая критические метаданные о всей файловой системе.

Содержание суперблока:

- Общее количество i-узлов (s_inodes_count)
- Общее количество блоков (s_blocks_count)
- Количество свободных блоков (s_free_blocks_count)
- Количество свободных i-узлов (s_free_inodes_count)
- Размер блока (s_block_size)
- Размер i-узла
- Время последнего монтирования (s_mtime)
- Время последней записи (s_wtime)
- Состояние файловой системы (clean или with errors)
- Сигнатура файловой системы (magic number)

Копии суперблока:

- Суперблок критично важен для ОС
- Копии хранятся в нескольких местах на диске
- Если основной суперблок повреждён, можно восстановить из копии
- fsck (file system check) использует эти копии для восстановления

39. Размещение файлов с использованием связанного списка

Принцип: каждый блок данных содержит указатель на следующий блок.

Структура:

- Первый блок файла указывается в i-узле
- Каждый блок содержит данные и указатель на следующий блок (или 0, если это последний)

Достоинства:

- Гибкость — можно использовать любые свободные блоки

- Экономия места в i-узле — не нужны все указатели

Недостатки:

- Медленный доступ к произвольной позиции (нужно идти по цепочке)
- Потеря указателя = потеря всего оставшегося файла
- Фрагментация может быть хуже

FAT (File Allocation Table):

- Таблица, где для каждого блока указано значение следующего блока
- Более надежно, чем указатели в блоках данных
- Позволяет быстро найти свободные блоки

40. Реализация каталогов

Простая реализация:

- Каталог — это файл с записями (name, inode_number)
- При поиске файла в каталоге линейный поиск по записям

Оптимизация:

- Хэш-таблица для быстрого поиска по имени
- Кэширование часто используемых записей (dcache в Linux)

Оптимизации при поиске в каталогах

1. **Кэширование (dcache)**
   - Кэш недавно открытых файлов в памяти
   - Избегает повторного чтения с диска
2. **Индексирование**
   - Создание индексов по часто используемым полям (имя)
3. **B-деревья**
   - Использование B-деревьев для организации записей в каталоге
   - Логарифмический поиск вместо линейного

41. Виртуальные файловые системы (VFS)

Определение: абстрактный слой в ядре, который предоставляет единый интерфейс для работы с разными физическими файловыми системами.

Назначение:

- Без VFS каждый open(), read() и т.д. должны знать о конкретной ФС
- С VFS приложения используют единый интерфейс независимо от ФС

Поддерживаемые файловые системы:

- Физические: ext2/3/4, btrfs, xfs, zfs, NTFS
- Виртуальные: tmpfs (в памяти), procfs (информация о процессах), devfs (устройства), sysfs (информация о системе), FUSE (пользовательские ФС)

Объекты VFS:

- `super_block` — описатель файловой системы
- `inode` — описатель файла/директории
- `dentry` — запись в директории (кэш пути)
- `file` — открытый файл (по дескриптору)

42. Оптимальный размер блока

Компромисс:

- Больший блок → меньше блоков для хранения одного файла, но больше внутренней фрагментации
- Меньший блок → больше блоков, но лучше использование места

Современные системы:

- 4096 байт (4KB) для ext4, большинства современных ФС
- Можно использовать большие блоки (8KB, 16KB, 64KB) для специальных приложений

Принципы отслеживания пустых блоков

1. **Битовая карта блоков (Block Bitmap)**
   - Каждый бит представляет один блок
   - 0 — свободно, 1 — занято
   - Быстрый поиск первого свободного блока
2. **Список свободных блоков (Free List)**
   - Связанный список свободных блоков
   - При выделении блока удалить из списка
3. **Счетчик свободных блоков**
   - Просто число доступных блоков
   - Быстрая проверка наличия места

43. Поддержание непротиворечивости файловой системы

Проблемы:

- Сбой питания может оставить ФС в неконсистентном состоянии
- Операции могут быть многошаговыми (изменить inode, обновить block bitmap)

Решения:

1. **Journaling (Журналирование)**
   - До выполнения операции записать в журнал, что будет сделано
   - После успешного завершения отметить в журнале
   - При восстановлении проверить журнал и завершить незавершённые операции
   - Использует ext3/4, btrfs, NTFS
2. **Copy-on-Write (CoW)**
   - Данные не изменяются на месте
   - Создается новая копия с изменениями
   - При сбое старые данные остаются целыми
   - Использует btrfs, zfs
3. **fsck (Filesystem Check)**
   - Проверка и восстановление при загрузке
   - Медленно (может занимать часы)
   - Современные ФС используют journal для ускорения

Увеличение производительности

1. **Кэширование блоков в памяти**
   - Buffer cache / page cache
   - Предотвращает частые обращения к диску
2. **Группирование блоков**
   - Попытка разместить файл в одной группе блоков
   - Уменьшает head movements на диске
3. **Отложенная запись (Delayed Writeback)**
   - Буферизировать записи и написать на диск периодически
   - Может объединить несколько операций в одно обращение

Отображение файлов на оперативную память

(см. раздел Memory Mapping)

44. Copy-On-Write (CoW)

Определение: техника оптимизации памяти, при которой несколько объектов первоначально ссылаются на одну копию данных, а копирование происходит только при попытке модификации.

Использование:

1. **При fork() в Unix:**
   - Дочерний процесс изначально совместно использует страницы памяти родителя
   - При записи в страницу создается копия только той страницы
   - Остальные страницы остаются общими
2. **В файловых системах:**
   - Снимки (snapshots) файловых систем могут использовать CoW
   - При изменении блока создается новый блок, старый остается для снимка

Преимущества:

- Экономия памяти при fork() — не требует копирования всей памяти
- Быстрое создание снимков

Реализация:

- При fork() используется механизм page protection
- Странице выставляются права только для чтения
- При попытке записи возникает page fault
- Ядро создает копию страницы и выставляет права для чтения-писи

45. Сетевое программирование

Понятие сокета

Сокет — это абстракция для сетевого взаимодействия, представляющая конечную точку связи.

Типы сокетов:

- SOCK_STREAM — TCP, надежная доставка (потоковый)
- SOCK_DGRAM — UDP, датаграммы (без гарантии доставки)
- SOCK_RAW — прямой доступ к протоколу

Домены (семейства):

- AF_INET — IPv4
- AF_INET6 — IPv6

- AF_UNIX — локальные сокеты (Unix domain sockets)

Именное пространство в сети

IP адреса:

- IPv4 — 4 байта (32 бита), записываются как xxx.xxx.xxx.xxx
- IPv6 — 16 байт (128 бит), записываются в шестнадцатеричной системе

Порты:

- 16-битные номера от 0 до 65535
- 0-1023 — зарезервированные (системные)
- 1024-49151 — зарегистрированные
- 49152-65535 — эфемерные (временные)

MAC адреса:

- 48-битные адреса на канальном уровне
- Уникальны в локальной сети

Виды установления соединения и передачи данных

1. **TCP (Transmission Control Protocol)**
   - Надежная доставка в порядке
   - Трёхсторонний handshake для установления соединения
   - Поток байтов (не сохраняются границы сообщений)
2. **UDP (User Datagram Protocol)**
   - Датаграммы (пакеты фиксированного размера)
   - Без гарантии доставки, возможно дублирование
   - Быстрее TCP, но менее надежно

46. Протоколы TCP и UDP

TCP (Transmission Control Protocol)

Характеристики:

- Connection-oriented — необходимо установить соединение
- Reliable — гарантирует доставку в правильном порядке
- Stream-oriented — данные это поток байтов
- Flow control — отправитель не переполняет приёмник
- Congestion control — реагирует на перегруженность сети

Установление соединения (3-way handshake):

1. Client отправляет SYN
2. Server отправляет SYN-ACK
3. Client отправляет ACK

Разрыв соединения (4-way handshake):

1. Client отправляет FIN
2. Server отправляет ACK
3. Server отправляет FIN
4. Client отправляет ACK

Достоинства:

- Надежность
- Порядок доставки гарантирован

Недостатки:

- Большие накладные расходы на установление соединения
- Медленнее UDP

UDP (User Datagram Protocol)

Характеристики:

- Connectionless — не требует установления соединения
- Unreliable — пакеты могут быть потеряны, дублированы, переупорядочены
- Datagram-oriented — каждый пакет самостоятельный
- Минимальные накладные расходы

Использование:

- DNS запросы
- Потоковое видео
- Онлайн игры
- VoIP

Достоинства:

- Быстрее TCP (нет handshake)
- Меньше накладных расходов
- Подходит для потоковых данных

Недостатки:

- Без гарантии доставки
- Приложение должно справляться с потерей пакетов

Порядок байтов (Byte Order)

Big-endian (Network byte order):

- Старший байт передается первым
- Используется в сетевых протоколах
- На Intel x86 используется little-endian, поэтому нужна конвертация

Little-endian:

- Младший байт передается первым
- Используется на x86/x64

Функции конвертации:

- `htons()` — host to network short
- `htonl()` — host to network long
- `ntohs()` — network to host short
- `ntohl()` — network to host long

47. Мультиплексирование сокетов

Определение: способность одного процесса одновременно работать с несколькими сокетами.

Использование fork для обработки клиентских соединений

Подход:

1. Основной процесс ждет подключения клиентов (accept)
2. При подключении нового клиента вызывается fork()
3. Дочерний процесс обслуживает этого клиента
4. Основной процесс продолжает ждать новых клиентов

Достоинства:

- Простая реализация
- Один клиент в одном процессе (с собственным адресным пространством)

Недостатки:

- Создание процесса затратно

- При большом количестве клиентов много процессов

Системные вызовы select, poll, epoll

select():

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

- Проверяет, готовы ли файловые дескрипторы для чтения/записи
- Блокирует до того, как хотя бы один готов
- Ограничение: максимум 1024 дескриптора (на большинстве систем)
- Работает на всех платформах

poll():

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

- Похож на select, но использует массив структур вместо битовых масок
- Нет жёсткого ограничения на количество дескрипторов
- Более удобен для большого количества дескрипторов

epoll(): (Linux-specific)

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

- Самый эффективный для большого количества дескрипторов
- O(1) сложность вместо O(n)
- Event-driven — уведомляет об изменениях
- Два режима: level-triggered и edge-triggered

48. Алгоритм инициализации сокета на сервере

Шаги:

```
// 1. Создать сокет
int server_sock = socket(AF_INET, SOCK_STREAM, 0);

// 2. Привязать адрес (bind)
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(server_sock, (struct sockaddr *)&addr, sizeof(addr));

// 3. Прослушивание (listen)
listen(server_sock, backlog);

// 4. Принятие соединений (accept)
struct sockaddr_in client_addr;
socklen_t client_addr_len = sizeof(client_addr);
int client_sock = accept(server_sock, (struct sockaddr *)&client_addr, &client_addr_len);

// 5. Обмен данными (send/recv)
char buffer[1024];
recv(client_sock, buffer, sizeof(buffer), 0);
send(client_sock, response, response_len, 0);

// 6. Закрытие
close(client_sock);
close(server_sock);
```

49. Алгоритм инициализации сокета на клиенте

Шаги:

```
// 1. Создать сокет
int client_sock = socket(AF_INET, SOCK_STREAM, 0);

// 2. Подключиться к серверу (connect)
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port);
inet_aton(server_ip, &server_addr.sin_addr);
connect(client_sock, (struct sockaddr *)&server_addr, sizeof(server_addr));

// 3. Отправка данных (send)
send(client_sock, data, data_len, 0);

// 4. Получение ответа (recv)
char buffer[1024];
recv(client_sock, buffer, sizeof(buffer), 0);

// 5. Закрытие
close(client_sock);
```

50. Алгоритм установления соединения и принятия соединения

Установление соединения (Client side)

Процесс connect():

1. Клиент отправляет SYN с портом назначения
2. Ядро создает полусоставленное соединение
3. Ядро ждет SYN-ACK от сервера
4. При получении SYN-ACK отправляет ACK
5. Соединение установлено, connect() возвращает управление

Принятие соединения (Server side)

Процесс accept():

1. Сервер вызывает listen() — переходит в режим прослушивания
2. При получении SYN от клиента ядро создает соединение в состоянии SYN_RCVD
3. Сервер вызывает accept() — получает сокет для этого клиента
4. Ядро отправляет SYN-ACK
5. При получении ACK от клиента соединение переходит в ESTABLISHED

Backlog в listen():

- Очередь полусоставленных соединений (SYN_RCVD)
- Очередь установленных соединений, ожидающих accept()
- Если очередь переполнится, новые SYN отклоняются

Заключение

Этот материал охватывает основные концепции операционных систем, изучаемые в курсе. Для полного понимания рекомендуется:

1. Изучить исходный код ОС (Linux, xv6)
2. Выполнить практические лабораторные работы
3. Экспериментировать с системными вызовами
4. Углубленное изучение конкретных компонентов