

1. Типы ОС (общего и специализированного назначения). Классификации ОС
2. Понятие и типы ресурсов. Понятие разделяемости, классификация разделяемых ресурсов.
3. Управление ресурсами. Функциональные компоненты ОС
4. Реентерабельный код, реализация, примеры применения
5. Системный вызов: назначение, реализация, примеры
6. Структуры и модели функционирования различных типов ядер.
7. Основные характеристики систем с различными типами архитектур. Предпочтительность той или иной архитектуры для различных классов ОС.
8. Модели функционирования в микроядерных архитектурах. Достоинства, недостатки
9. Многозадачность
10. Мультипроцессирование
11. Многозадачность при мультипроцессировании. Проблемы реализации кода.
12. Интерфейсы программирования и пользователя. Стандартизация интерфейсов
13. API – назначение, структура, проблемы совместимости, реализация
14. Стандарты POSIX, стандарты API различных ОС
15. Архитектурный слой middleware
16. Организация файлов, типы, форматы
17. Виртуальная ФС. Структуры vfs в Unix
18. Методы размещения и доступа к файлам
19. Методы учета занятого/свободного пространства (ФС)
20. Взаимосвязь методов размещения, доступа и учета свободного пространства (ФС)
21. Жизненный цикл и состояния процесса
22. Контекст и дескриптор процесса на примере Unix
23. Структуры контекстов процессов и нитей
24. Переключение контекста и затраты ресурсов на переключение
25. Методы и алгоритмы планирования

26. Инверсия приоритетов

27. Методы IPC в Unix-подобных ОС: состав и общая характеристика каждого типа IPC (включая сетевые)

28. Обработка событий

29. Синхронизация при разделении ресурсов, средства синхронизации

30. Типы памяти. Виды организации виртуальной памяти

31. Управление памятью, распределенной динамическими разделами

32. Механизмы преобразования виртуального адреса в физический при различных организациях памяти

33. Базовые сервисы диспетчера виртуальной памяти

34. Принцип кэширования данных. Стратегия подкачки страниц. Свопинг.

35. Совместное использование памяти процессами.

36. Защита памяти. Средства преобразования. Распределение памяти между системными и прикладными задачами

37. Состав, функции, структура системы ввода/вывода. BIOS, EFI/UEFI.

38. Синхронный и асинхронный ввод/вывод.

39. Способы обмена с внешними устройствами, их программная реализация. Драйверы.

40. Иерархия драйверов. Многослойная модель драйвера.

41. Унифицированная модель драйвера

42. Обработка прерываний, исключений, ловушек. Типы прерываний.

43. Обработчики аппаратных прерываний, правила написания программного кода обработчиков

44. API-поддержка обработки прерываний и событий для проектирования приложений

45. Современные средства разработки ПО (SDK, IDE, Toolkits).

Инструментальные средства ОС для трассировки, анализа и отладки кода.

# 1. Типы ОС (общего и специализированного назначения). Классификации ОС.

**ОС компьютера** – комплекс взаимосвязанных управляющих и обрабатывающих программ, который действует как интерфейс между приложениями/пользователем с одной стороны и аппаратурой компьютера с другой.

## **ОС общего назначения:**

1. **Универсальность:** разработаны для широкого спектра задач и приложений.
2. **Пользовательский интерфейс:** графический интерфейс пользователя (GUI) и удобные средства взаимодействия с компьютером.
3. **Расширяемость:** возможность установки и запуска сторонних приложений и программного обеспечения.
4. **Общедоступность:** широко доступны для обычных пользователей и могут быть установлены на различные устройства.

## **ОС специализированного назначения:**

1. **Специализация:** разработаны для конкретных задач или областей применения.
2. **Оптимизация:** нацелены на оптимизацию производительности и решение узкоспециализированных задач.
3. **Управление ресурсами:** предоставляют дополнительные функции и возможности для управления и контроля конкретными ресурсами.
4. **Ограниченность:** могут быть ограничены в функциональности и поддержке сторонних приложений.

## **Классификация ОС:**

### ! по назначению:

- ~ универсальные (общего назначения)
- ~ специализированные (фиксированный набор приложений)

### ! по способу загрузки:

- ~ загружаемые (выполняют функции после считывания с некоторого накопителя)
- ~ резидентные (постоянно загружены в память, чаще специализированные)

### ! в соответствии с особенностями управления ресурсами:

- ~ по управлению процессорным временем: наличие многозадачности:
  - ~~ однозадачные (1 user, 1 задача)
  - ~~ многозадачные (много users, много задач)
- ~ по видам многозадачности
  - ~~ не вытесняющие (процесс выполняется до того, пока сам не освободит ресурс/завершится)
  - ~~ вытесняющие (ОС распределяет кванты времени для всех задач, то есть может быть прервана)

### ! поддержка работы пользователя:

- ~ однопользовательские
- ~ многопользовательские

### ! в соответствии с алгоритмами управления процессором:

- ~ многозадачные/однозадачные/мультипрограммирование

~ многонитевые/многопоточные и без

~ многопроцессорные/однопроцессорные/мультипроцессирование

## 2. Понятие и типы ресурсов. Понятие разделяемости, классификация разделяемых ресурсов.

### Понятие ресурсов

все, чем ОС может управлять и распределять между процессами и пользователями.

### Типы ресурсов

- Разделяемые
  - ✓ Одновременное использование (использование несколькими потребителями в один момент времени)
  - ✓ Параллельное использование (в течении некоторого времени потребители используют ресурс попеременно)

- Неразделяемые

### Виды ресурсов

- Аппаратные
  - ✓ Процессорное время (процесс или набор процессов)
  - ✓ Оперативная память
    - Основные способы разделения оперативной памяти:
      - Временной
      - Пространственный
    - Способы выделения памяти процессорам:
      - Статический (резервация для фиксированного числа процессоров)
      - Динамический (по запросу процессора)
  - ✓ Внешняя память. 2 ресурса:
    - Пространство (разделяется пространственным способом)
    - Доступ (разделяется временным способом)
  - ✓ Внешние устройства:
    - Параллельно разделяемые (прямой доступ) | разделяемый ресурс
    - Последовательный доступ | неразделяемый ресурс
- Программные
  - Основные программные ресурсы – системные программные модули, разделяемые между выполняющимися процессами/программами/задачами.
  - Разделяемыми могут быть только те, для которых возможно многократное использование без искажения кода и данных:
    - ✓ Реентабельные (могут быть использованы несколькими потоками или процессами одновременно без взаимного влияния)
    - ✓ Повторно-входимые (могут быть использованы несколькими потоками одновременно, но для этого нужна синхронизация доступа к ресурсам)
- Информационные (данные):
  - ✓ Доступ по чтению без спец средств
  - ✓ Доступ с возможностью изменения информационных ресурсов требует специальных алгоритмов



### 3. Управление ресурсами. Функциональные компоненты ОС.

Независимо от типа ресурсов **управление ресурсами осуществляется с помощью:**

- 1) Планирование ресурса;
- 2) Удовлетворение запросов на ресурсы
- 3) Отслеживание и учет состояния ресурса
- 4) Разрешение конфликтов между потребителями ресурсов

#### **Функциональные компоненты локальной ОС**

Основное представление состава ОС – группирование по функциональному признаку в соответствии с **типами локальных ресурсов:**

- 1) подсистема управления процессами. Планировщик процессов определяет порядок и приоритет выполнения процессов на ЦП.
- 2) подсистема управления памятью. Диспетчер памяти контролирует выделение и освобождение ресурсов памяти.
- 3) ФС. Файловая система управляет хранением и извлечением файлов на дополнительных устройствах хранения.
- 4) система ввода/вывода. координирует поток данных между ЦП, памятью и периферийными устройствами.
- 5) UI/GUI. позволяет пользователям вводить данные, управлять приложениями и отображать информацию с помощью графических элементов.
- 6) Программные средства защита данных и администрирования. Они предназначены для обеспечения безопасности данных и управления компьютерной средой на уровне отдельного ПК.
- 7) Middleware
- 8) Множественные среды
- 9) Системы и среды программирования
- 10) Платформы и среды разработки
- 11) Виртуальные машины

#### 4. Реентерабельный код, реализация, примеры применения.

**Реентерабельный код** - это многократно используемая процедура, которую несколько программ могут вызывать, прерывать и повторно вызывать одновременно.

Обеспечение реентерабельности является ключевым моментом при программировании многозадачных систем, в частности, операционных систем.

##### **Правила для реентерабельности:**

- 1) Реентерабельный код может не содержать никаких статических или глобальных непостоянных данных без синхронизации;
- 2) Реентерабельный код не может изменять сам себя без синхронизации;
- 3) Реентерабельный код не может вызывать нереентерабельные программы или подпрограммы.

##### **\* Примечание**

Функция повторного входа может потребоваться более чем одной задачей, не опасаясь повреждения данных.

Функция повторного входа может быть прервана в любое время и возоблена позже, без потери данных.

Реентерабельные функции используют либо локальные переменные, либо защищают данные при использовании глобальных переменных.

Пример функции повторного входа:

```
void strcpy(char *dest, char *src)
{
    while (*dest++ = *src++);

    *dest = NULL;
}
```

Здесь копии фрагментов strcpy помещены в стек задачи, strcpy может быть вызвана несколькими задачами, не опасаясь, что задачи будут искажать указатели друг друга.

Пример нереентерабельной функции:

```
int Temp;
void swap(int *X, int *Y)
{
    Temp = *X;
    *X = *Y;
    *Y = Temp;
}
```

Функция swap меняет содержимое двух аргументов.

Условия применения:

- Вытесняющее ядро
- Прерывания включены
- Temp объявлена как глобальное целое число

##### **Преобразование кода в реентерабельный**

Различные способы сделать код из приведенного примера реентерабельным:

- Объявить Temp локальной для swap.
- Отключить прерывания перед операцией и включить их позже.
- Использовать семафор.



## 5. Системный вызов: назначение, реализация, примеры.

**Системный вызов** является механизмом, который обеспечивает интерфейс между процессом и операционной системой. Это программный метод, при котором компьютерная программа запрашивает сервис у ядра ОС.

Системные вызовы создают, удаляют и используют различные объекты, главные из которых – процессы и файлы. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют *прерывание процессора*, после чего управление передается *обработчику данного вызова*, входящему в ядро операционной системы.

**Параметры могут передаваться через:**

- **Регистры:** сохраняются в регистрах процессора.
- **Память:** функция получает указатель на блок памяти с параметрами. Такая реализация в Линукс.
- **Стек:** функция берёт параметры из стека. Такая реализация в Windows.

В отличие от подпрограмм, при системном вызове задача переходит в привилегированный режим или режим ядра. Отсюда другое название системных вызовов – программные прерывания.

В kernel mode работает код ядра ОС, исполняется он в адресном пространстве и в контексте вызвавшей его задачи. Это значит, что ядро ОС имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса  $\geq 1$  областей памяти с параметрами вызова и адреса  $\geq 1$  областей памяти для результатов вызова.

В большинстве ОС системный вызов осуществляется командой программного прерывания. Программное прерывание – это синхронное событие, оно может быть повторено при выполнении одного и того же программного кода.

Существует таблица указателей на функции, реализующие системные вызовы.

**Архитектура системного вызова:**

**Шаг 1)** Процессы выполняются в пользовательском режиме до тех пор, пока системный вызов не прервет его.

**Шаг 2)** Ядру передаётся номер системного вызова. После этого системный вызов выполняется в kernel mode в приоритетном порядке.

**Шаг 3)** По завершении выполнения системного вызова управление возвращается в user mode.

**Типы системных вызовов:**

- Управление процессами
- Управление файлами
- Управление устройствами
- Управление памятью
- Коммуникация

**Примеры, когда нужны системные вызовы:**

- Чтение и запись из файлов.
- Если файловая система хочет создать или удалить файлы
- Создание и управления новыми процессами.
- Сетевые подключения требуют системных вызовов для отправки и получения пакетов.

- Для доступа к аппаратным устройствам, таким как сканер, принтер и т.д.

Примеры функций системных вызовов, которые мы использовали в лабораторных работах на парах:

- `wait()` – приостановка родительского процесса .
- `fork()` – создание процессов, которые являются их копиями.
- `exec()` – замена старого файла или программы из процесса новым файлом или программой.
- `kill()` – отправляет сигналы.
- `exit()` – завершение программы.

## 6. Структуры и модели функционирования различных типов ядер.

- **Монолитное ядро.** Все части ядра работают в одном адресном пространстве. Все компоненты ее ядра являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур.

**Достоинства:** Незначительно выше скорость работы, упрощенная разработка модулей.

**Недостатки:** Поскольку всё ядро работает в одном адресном пространстве, сбой в одном из компонентов может нарушить работоспособность всей системы. Требуется перекомпиляция ядра.

- **Модульное ядро.** В отличие от «классических» монолитных ядер, модульные ядра, как правило, не требуют полной перекомпиляции ядра при изменении состава аппаратного обеспечения компьютера. Вместо этого модульные ядра предоставляют тот или иной механизм подгрузки модулей ядра, поддерживающих то или иное аппаратное обеспечение (например, драйверов). При этом подгрузка модулей может быть, как динамической, так и статической.

- **Микроядро** предоставляет только элементарные функции управления процессами и минимальный набор абстракций для работы с оборудованием. Большая часть работы осуществляется с помощью специальных пользовательских процессов, называемых *сервисами*. Решающим критерием «микроядерности» является размещение всех или почти всех драйверов и модулей в сервисных процессах, иногда с явной невозможностью загрузки любых модулей расширения в собственное микроядро, а также разработки таких расширений.

**Достоинства:** Устойчивость к сбоям оборудования, ошибкам в компонентах системы. Основное достоинство микроядерной архитектуры — высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

**Недостатки:** Передача данных между процессами требует накладных расходов.

- **Экзоядро** — ядро операционной системы, предоставляющее лишь функции для взаимодействия между процессами, безопасного выделения и освобождения ресурсов. Предполагается, что API для прикладных программ будут предоставляться внешними по отношению к ядру библиотеками (откуда и название архитектуры).

Возможность доступа к устройствам на уровне контроллеров позволит эффективно решать некоторые задачи, которые плохо вписываются в рамки универсальной ОС.

- **Наноядро** — архитектура ядра операционной системы, в рамках которой крайне упрощённое и минималистичное ядро выполняет лишь одну задачу — обработку аппаратных прерываний, генерируемых устройствами компьютера. После обработки прерываний от аппаратуры наноядро, в свою очередь, посылает информацию о результатах обработки вышележащему программному обеспечению при помощи того же механизма прерываний.

- **Гибридные ядра** — это модифицированные микроядра, позволяющие для ускорения работы запускать «несущественные» части в пространстве ядра.

## 7. Основные характеристики систем с различными типами архитектур. Предпочтительность той или иной архитектуры для различных классов ОС.

**Архитектура ОС** - структурная и функциональная организация операционной системы на основе некоторых программных модулей.

### **Типы архитектур:**

**Монолитная.** Единая неструктурированная система программных модулей. В ней используется одно ядро, которое компонуется как одна программа, работающая в привилегированном режиме. Таким образом, все её компоненты являются составными частями одной программы, используют общие структуры данных.

#### Достоинства:

- 1) Относительно высокая производительность, так как все компоненты ядра общаются в одном адресном пространстве;
- 2) Простота добавления новых функций в ядро ОС, так как это делается путем добавления в уже существующий модуль;

#### Недостатки:

- 1) Поскольку всё ядро работает в одном адресном пространстве, сбой в одном из компонентов может нарушить работоспособность всей системы;
- 2) Невозможность модификации компонентов без перекомпиляции всего ядра;
- 3) Тяжело переносить на другие аппаратные платформы;

**Использование:** Монолитная архитектура ОС может использоваться в операционных системах для персональных компьютеров и серверов. Она предпочтительна для ОС, которые должны обеспечивать высокую производительность и имеют статический набор функций.

**Многоуровневая.** Структурированная система, состоящая из нескольких уровней абстракции, каждый из которых выполняет определённые функции. Каждый уровень взаимодействует только с соседним нижележащим через его интерфейс. Внутренние структуры данных каждого уровня не доступны другим уровням, а реализации процедур уровня скрыты и не зависят от реализаций процедур внутри других уровней.

#### Достоинства:

- 1) Вносить функциональность достаточно в один уровень, не затрагивая остальные;
- 2) При переносе на другую платформу меняется только аппаратный уровень;
- 3) Можно достаточно легко выявить уровень, на котором произошёл сбой; компоненты одного уровня не влияют на другой.

#### Недостатки:

- 1) Сложны к разработке (тяжело спроектировать такую систему и правильно определить порядок слоев)
- 2) Эффективность замедлена из-за общения между слоями, что требует больше вычислительных ресурсов

**Использование:** Многоуровневая архитектура операционной системы (ОС) предпочтительна для крупных и сложных систем, где высокая модульность и расширяемость являются важными требованиями.

**Микроядерная (или архитектура клиент-сервер).** Ядро ОС содержит только самые основные функции, а все остальные функции выполняют процессы-сервера в пользовательском пространстве. Большинство составляющих ОС являются

самостоятельными программами, взаимодействие между которыми осуществляется через микроядро.

Достоинства:

- 1) Система становится более гибкой из-за минимальной функциональности ядра.
- 2) Устойчивость к ошибкам: благодаря тому, что компоненты ОС выполняются в отдельных процессах, ошибки в одной из них не влияют на работу другой;
- 3) Благодаря тому, что компоненты-сервера работают в пользовательском пространстве, система становится более безопасной;

Недостатки:

- 1) Передача данных между процессами снижает производительность;

Использование: Микроядерная архитектура ОС предпочтительна для систем, где безопасность, надежность и модульность критически важны, например, для ОС, работающих в автоматизированных системах управления производством.

## 8. Модели функционирования в микроядерных архитектурах. Достоинства, недостатки

Суть микроядерной архитектуры состоит в следующем. В привилегированном режиме остается работать только очень небольшая часть ОС, называемая **микро-ядром**. Микроядро защищено от остальных частей ОС и приложений.

В микроядерной архитектуре функциональность операционной системы разбивается на **отдельные модули**, которые работают независимо друг от друга.

Работа микроядерной операционной системы соответствует известной модели **клиент-сервер**, в которой роль транспортных средств выполняет микроядро.

### Достоинства

- Высокая степень переносимости обеспечивается тем, что весь машинно-зависимый код изолирован в микроядре, поэтому для переноса системы на новый процессор требуется меньше изменений и все они логически сгруппированы вместе.
- Расширяемость. Микроядерная архитектура позволяет не только добавлять, но и сокращать число компонентов ОС.
- надежность ОС. Каждый сервер выполняется в виде своего собственного процесса в своей собственной области памяти. Так как серверы выполняются в пользовательском режиме, они не имеют непосредственного доступа к аппаратуре и не могут модифицировать память, в которой хранится микроядро.

### Недостатки

- При классической организации ОС выполнение системного вызова - двумя переключениями режимов, а при микроядерной - четырьмя. микроядро при прочих равных условиях всегда будет менее производительной.
- Главная проблема - решение, какие функции ОС необходимо локализовать в микроядре, а какие вынести в серверные приложения пользовательского уровня.

## 9. Многозадачность

**Многозадачность** — способность операционной системы поддерживать одновременное совместное выполнение и взаимодействие нескольких задач. При этом задачи делят между собой общие ресурсы.

### Виды многозадачности:

- **Процессная многозадачность.**

программа — наименьший элемент управляемого кода, которым может управлять планировщик операционной системы. Многозадачная система позволяет двум или более программам выполняться одновременно.

- **Поточная многозадачность.**

Многопоточность — специализированная форма многозадачности. Наименьший элемент управляемого кода — поток. Многопоточная система предоставляет возможность одновременного выполнения одной программой двух и более задач. (доп. информация: каждый поток имеет свой стек вызовов, но разделяет с другими потоками память и ресурсы процесса).

Также многозадачность **бывает параллельной и псевдопараллельной**:

- По-настоящему **параллельное** исполнение процессов возможно только в многопроцессорной системе.

- В однопроцессорной многозадачной системе поддерживается так называемое **псевдопараллельное исполнение**, при котором создается видимость параллельной работы нескольких процессов. В таких системах, процессы выполняются последовательно, занимая малые кванты процессорного времени.

**Типы псевдопараллельной многозадачности** (наверное, это можно не писать):

- **Вытесняющая многозадачность** - это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

- **Невытесняющая многозадачность** - это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

## 10. Мультипроцессирование

**Мультипроцессирование** – одновременное выполнение нескольких процессов на вычислительной машине, имеющих несколько процессоров; если количество одновременно выполняемых процессов не превышает количества процессоров, то все процессы будут одновременно выполняться по времени, каждый на своем процессоре. Если больше, то на некоторых процессорах будет выполняться мультипрограммирование.

**Мультипрограммирование** – одновременное выполнение нескольких процессов на вычислительной машине с одним процессором. В этом случае распараллеливание процессов осуществляется за счет того, что пока для одного процесса выполняется операция ввода-вывода, другой процесс использует время ЦП.

При рассмотрении вопросов мультипроцессирования часто используют такие термины, как *симметричный* и *асимметричный* мультипроцесс (м/п) (применительно к аппаратуре и к организации вычислительного процесса).

- *Симметричная* мультипроцессорная архитектура представляет из себя мультипроцессорную систему с несколькими одинаковыми м/п. Обычно такие м/п организованы в одном корпусе.

- В случае *асимметричной* мультипроцессорной архитектуры вычислительная система содержит несколько различных процессоров, выполняющих различные функции. В таких системах 1 из процессоров управляет работой остальных и называется ведущим, все остальные – ведомыми.

При асимметричной архитектуре возможно только асимметричная организация вычислительного процесса, а при симметричной архитектуре – как симметричная, так и асимметричная организация вычислительного процесса.



## **11. Многозадачность при мультипроцессировании. Проблемы реализации кода.**

**Мультипроцессорная обработка** - это способ организации вычислительного процесса, при котором несколько задач могут одновременно параллельно выполняться на разных обрабатывающих устройствах системы. В качестве вычислителей в современных компьютерных системах могут быть процессоры или ядра одного процессора или нескольких процессоров.

Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами. Многопроцессорные системы требуют от операционной системы особой организации, с помощью которой сама ОС, а также поддерживаемые ею приложения, могли бы выполняться параллельно отдельными вычислителями системы.

**Способы организации вычислительного процесса, реализуемые той или иной ОС при мультипроцессировании, могут быть симметричными и асимметричными.**

**Асимметричное** мультипроцессирование по принципу "ведущий-ведомый" является наиболее простым способом организации. Он предполагает выделение одного вычислителя в качестве ведущего, на котором работает вся ОС и который управляет остальными ведомыми процессами, выполняет функции распределения прикладных задач и ресурсов.

ОС, использующая **симметричное** мультипроцессирование, полностью децентрализована. Разные вычислители могут одновременно обслуживать как разные, так и одинаковые модули общей ОС, для этого все программы ОС должны быть реентерабельными.

## 12. Интерфейсы программирования и пользователя. Стандартизация интерфейсов

**Интерфейс (И.)** – совокупность средств информационного взаимодействия двух объектов или процессов. Взаимодействие – обмен информацией и управление. Обычно И. несимметричен: одна его сторона – управляющая (master, client), другая – подчиненная (slave, server). **четыре вида интерфейса:**

**1. Аппаратный** – между цифровыми устройствами. Обеспечивает двусторонний обмен сигналами на физическом уровне. Средства: кабели, шины, разъемы, сигналы, алгоритмы, временные диаграммы. Примеры:

**2. Аппаратно-программный И.** – между программой и аппаратурой.

**3. Межпрограммный И.** – между компонентами или слоями программного обеспечения. Традиционное название: CALL-интерфейс (набор форматов вызова процедур; в СУБД – CLI (Call Level Interface)), современное название – API (Application Programming Interface).

**4. UI** – И. пользователя – между пользователем и программой. Диалоговое взаимодействие с помощью клавиатуры, мыши, графических элементов на экране и пр.

### **Принципы хорошего И.:**

- **Инкапсуляция** внутреннего устройства, изоляция сторон И. Сервер не должен знать, **зачем** он задействуется, клиент – **как** тот действует.
- **Экономность, лаконичность И.** – минимум обмениваемой информации.
- **Стандартизация, унификация И.** – способствует сопрягаемости (interoperability) и повторному использованию компонентов.

**Стандартизация** — принятие соглашения по спецификации, производству и использованию аппаратных и программных средств вычислительной техники; установление и применение стандартов, норм, правил и т.п.

### 13. API – назначение, структура, проблемы совместимости, реализация

**API** (от англ. Application Programming Interface) — описание способов взаимодействия одной компьютерной программы с другими.

#### Варианты реализации API

##### 1. Реализация функций API на уровне ОС.

Ответственность за выполнение функций API несет ОС. Программа, выполняющая функции, либо непосредственно входит в состав ОС, либо поставляется в составе динамически загружаемых библиотек.

Достигается **наибольшая эффективность выполнения функций API** (т.к. прикладная программа обращается непосредственно к ОС).

Недостаток: **практически полное отсутствие переносимости** не только кода результирующей программы, но и кода исходной программы.

##### 2. Реализация функций API на уровне системы программирования.

Функции API предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Чаще всего это библиотека времени исполнения – RTL (run time library).

Эффективность будет несколько ниже, чем при непосредственном обращении к функциям ОС, т.к. библиотечная функция все равно выполняет обращения к функциям ОС. Переносимость исходной программы будет самой высокой, т.к. синтаксис и семантика всех функций будут строго регламентированы в стандарте соответствующего языка программирования. Для каждой ОС будет требоваться свой код RTL

##### 3. Реализация функций API с помощью внешних библиотек.

Функции API предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком

Эффективность наименьшая, т.к. внешняя библиотека обращается как к функциям ОС, так и к функциям RTL.

**Соответственно, API отвечает на вопрос “Как ко мне, к моей системе можно обратиться?”, и включает в себя:**

- саму операцию, которую мы можем выполнить,
- данные, которые поступают на вход,
- данные, которые оказываются на выходе (контент данных или сообщение об ошибке).

#### Совместимость на уровне двоичного кода

возможность потребителя API использовать его в более новой версии без перекомпиляции. Такие изменения, как добавление методов или добавление новой реализации интерфейса к типу, не влияют на совместимость двоичных файлов.

#### Совместимость исходного кода

возможность перекомпиляции существующих потребителей API без изменения исходного кода. Изменение, несовместимое на уровне исходного кода, имеет место, когда потребителю необходимо изменить исходный код для его успешной сборки в более новой версии API.

### **Совместимость во время разработки**

сохранение возможностей разработки в разных версиях Visual Studio и других средах разработки. однако самый важный аспект совместимости во время разработки — совместимость проекта. Необходимо, чтобы проект или решение можно было открывать и использовать в более новой версии среды разработки.

### **Обратная совместимость**

возможность выполнения существующего потребителя API в новой версии без изменения поведения. На обратную совместимость влияют как изменения поведения, так и изменения в совместимости на уровне двоичного кода. Если потребитель не может работать с более новой версией API или ведет себя иначе при ее использовании, этот API не обладает обратной совместимостью.

### **Прямая совместимость**

возможность выполнения существующего потребителя API в предыдущей версии без изменения поведения. Если потребитель не может работать с более старой версией API или ведет себя иначе при ее использовании, этот API не обладает прямой совместимостью.

## 14. Стандарты POSIX, стандарты API различных ОС

**POSIX** (англ. *Portable Operating System Interface* — переносимый интерфейс операционных систем) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода.

**Стандарт состоит из четырех основных разделов:**

- **Основные определения** — список основных определений и соглашений, используемых в спецификациях, и список заголовочных файлов языка Си, которые должны быть предоставлены соответствующей стандарту системой.
- **Оболочка и утилиты** — описание утилит и командной оболочки sh, стандарты регулярных выражений.
- **Системные интерфейсы** — список системных вызовов языка Си.
- **Обоснование** — объяснение принципов, используемых в стандарте.

### POSIX.1. Системное API для языка Си

- Создание и управление процессами
- Сигналы
  - ✓ Исключения плавающей точки
  - ✓ Нарушение сегментации
  - ✓ Запрещённые директивы
  - ✓ Ошибки шины
  - ✓ Таймеры
- Операции над файлами (каталогами)
- Каналы
- Библиотека Си (стандарт Си)
- Интерфейс и контроль ввода-вывода
- Триггеры процессов

### POSIX.2. Командная оболочка и утилиты

- Командный интерпретатор
- Программные утилиты

### POSIX.1b. Расширения реального времени

- Планировка приоритетов
- Сигналы реального времени
- Часы и таймеры
- Семафоры
- Передача сообщений
- Управление памятью
- Синхронизация файлов и асинхронный ввод-вывод
- Интерфейс блокировки виртуальной памяти

### POSIX.1c. Поток

- Создание, контроль и завершение выполнения потоков

- Планировщик потоков
- Синхронизация потоков
- Обработка сигналов

## **POSIX.1-1996**

Объединил все предыдущие (кроме POSIX 2.

## **POSIX.1d. Дополнительные расширения реального времени**

## **POSIX.1g. Протоколо-независимые интерфейсы**

## **POSIX.1j. Продвинутое расширение реального времени**

**API** ( от англ. *Application Programming Interface*<sup>[1]</sup>) — описание способов взаимодействия одной компьютерной программы с другими.

API (интерфейс прикладного программирования) упрощает процесс программирования при создании приложений, абстрагируя базовую реализацию и предоставляя только объекты или действия, необходимые разработчику.

## **Windows API**

*Windows API* спроектирован для использования в языке Си для написания прикладных программ, предназначенных для работы под управлением операционной системы MS Windows.

Версии

- **Win16** — первая версия *WinAPI* для 16-разрядных версий *Windows*.
- **Win32** — 32-разрядный *API* для современных версий *Windows*.
- **Win32s** — подмножество *Win32*, устанавливаемое на семейство 16-разрядных систем *Windows 3.x* и реализующее ограниченный набор функций *Win32* для этих систем.
- **Win64** — 64-разрядная версия *Win32*, содержащая дополнительные функции *Windows* на платформах *x86-64* и *IA-64*.

## 15. Архитектурный слой middleware

**Сервис ПО промежуточного слоя (middleware service)** - сервис общего назначения, который располагается между платформами и приложениями. Под платформой мы подразумеваем набор низкоуровневых сервисов и функциональные особенности, определяемые парой: архитектура процессора и API операционной системы. Оно обеспечивает функциональность для интеллектуального и эффективного взаимодействия приложений, содействуя ускорению внедрения инноваций.

Промежуточное ПО действует как мост между различными технологиями, инструментами и базами данных, содействуя интеграции их в единую систему без дополнительных усилий. Затем единая система предоставляет своим пользователям унифицированные услуги. Например, клиентское приложение в среде Windows отправляет и получает данные с сервера Linux, но пользователи приложения не замечают эту разницу операционных систем.

Сервисы middleware представляют приложениям разнообразные функции API, которые, в сравнении с функциями операционных систем и сетевых служб, **обеспечивают:**

- прозрачный доступ к другим сетевым сервисам и приложениям;
- независимость от других сетевых сервисов;
- высокую надежность и постоянную готовность.

### **Виды промежуточного ПО:**

- Программное обеспечение для межпрограммного взаимодействия (см. также IPC Inter-Process Communication).
- Программное обеспечение доступа к базам данных.

## 16. Организация файлов, типы, форматы

**Файл** - именованная совокупность данных, способная долговременно храниться во внешней памяти, доступная пользователям/приложениям для чтения/записи/запуска.

### Идентификация файлов:

- Имя
- Дескриптор
  - ✓ Индексный, i-node. Просто номер файла в ФС
  - ✓ Файловый, fd. Идентификатор процесса, возвращается например при open()

Принципы размещения файлов, каталогов и системной информации на реальном носителе описываются физ.организацией ФС.

В Unix-подобных системах используется такая концепция, как монтирование файловых систем. Это позволяет подключать новые устройства (например, USB-накопители) к уже существующему дереву каталогов.

**Каждый файл в Unix-подобной системе имеет набор атрибутов, которые включают:**

- Владелец файла
- Права доступа
- Тип файла

Тип файла можно узнать командой `ls -l`.

### Файлы системы могут быть следующих типов:

• Обычный файл. Это наиболее общий тип файла, который может содержать любые данные. Файлы могут быть текстовыми или бинарными.

**b** Специальный файл блочного устройства. Представляют физические устройства с блочным доступом (жесткие диски и CD-ROM).

**c** Файл символьного устройства. Принтеры, клавиатура, мышь.

**d** Директория. Представляют собой контейнеры для других файлов и каталогов.

**l** Ссылка.

- ✓ **Символьная:** содержимое файла - не данные, а путь к другому файлу. Удалять ссылку и файл можно независимо.

- ✓ **Жёсткая:** другое имя, но i-node тот же.

**p** FIFO. Канал для взаимодействия процессов.

**s** Сокет. Используются для обмена данными между процессами, возможно, работающими на разных машинах.

### Форматы файлов

• **Текстовые файлы** - Это файлы, данные в которых представлены в виде текста. Файлы исходного кода, скрипты, конфигурационные файлы и документы.

• **Бинарные файлы** - Это файлы, данные в которых представлены в бинарном формате. Исполняемые файлы, изображения, аудио и видео файлы, и многие другие типы файлов.



## 17. Виртуальная ФС. Структуры `vfs` в Unix

различные типы файловых систем существенно отличаются по внутренней архитектуре. В то же время современные версии ОС UNIX обеспечивают одновременную работу с несколькими типами файловых систем. Среди них можно выделить локальные файловые системы различной архитектуры, удаленные и даже отличные от файловой системы UNIX. Такое сосуществование обеспечивается путем разделения каждой файловой системы на зависимый и независимый от реализации уровни, последний из которых является общим и представляет для остальных подметем ядра некоторую абстрактную файловую систему. Независимый уровень также называется виртуальной файловой системой.

### Виртуальный индексный дескриптор

Дисковый файл обычно имеет связанную с ним структуру данных, называемую метаданными или индексным дескриптором, где хранятся основные характеристики данного файла и с помощью которой обеспечивается доступ к его данным. Интерфейсом работы с файлами является виртуальный индексный дескриптор `vnode` (от *virtual inode*).

Метаданные всех открытых файлов (файлов, на которые ссылаются один или более процессов) представлены индексными дескрипторами в памяти. В качестве последних в виртуальной файловой системе выступают виртуальные дескрипторы `vnode`. Структура данных виртуального индексного дескриптора одинакова для всех файлов, независимо от типа реальной файловой системы, где фактически располагается файл. Данные такого дескриптора содержат информацию, необходимую для работы виртуальной файловой системы, а также неизменные характеристики файла, такие как, например, тип файла.

### Монтирование файловой системы

Работа с файлами невозможна, пока соответствующая файловая система не встроена в существующее иерархическое дерево. Только после этого ядро сможет выполнять файловые операции, такие как создание, открытие, чтение или запись в файл. Эта операция встраивания получила название подключения или монтирования файловой системы.

Каждая подключенная файловая система представлена на независимом уровне в виде структуры `vfs`, аналоге записи таблицы монтирования дисковой файловой системы. Структуры `vfs` всех подключенных файловых систем организованы в виде односвязного списка, в совокупности обеспечивая информацию, необходимую для обслуживания всего иерархического дерева, а также информацию о реальной файловой системе, которые не изменяются на протяжении работы. Первой записью списка всегда является корневая файловая система.

Для инициализации и монтирования реальной файловой системы в UNIX имеется коммутатор (переключатель) файловых систем, адресующий процедурный интерфейс для каждого типа файловой системы, поддерживаемой ядром.

## 18. Методы размещения и доступа к файлам

### Методы размещения файлов.

#### Непрерывное размещение.

Простейшая схема размещения заключается в хранении каждого файла на диске в виде непрерывной последовательности блоков.

Следует заметить, что каждый файл начинается от границы нового блока, поэтому,

#### *Преимущества:*

- Просто реализовать, поскольку отслеживание местонахождения принадлежащих файлу блоков сводится всего лишь к запоминанию двух чисел: дискового адреса первого блока и количества блоков в файле.

- Превосходная производительность считывания, поскольку весь файл может быть считан с диска за одну операцию. Для нее потребуется только одна операция позиционирования (на первый блок).

#### *Недостатки:*

- Со временем диск становится фрагментированным. При удалении файла его блоки освобождаются и на диске остается последовательность свободных блоков.

#### Размещение с использованием связанного списка

представлении каждого файла в виде связанного списка дисковых блоков. Первое слово каждого блока используется в качестве указателя на следующий блок, а вся остальная часть блока предназначена для хранения данных.

#### *Преимущества:*

- Может быть использован каждый дисковый блок
- Достаточно, чтобы в записи каталога хранился только дисковый адрес первого блока. Всю остальную информацию можно найти, начиная с этого блока.

#### *Недостатки:*

- Произвольный доступ является слишком медленным. Чтобы добраться до блока  $n$ , операционной системе нужно начать со стартовой позиции и прочитать поочередно  $n - 1$  предшествующих блоков.

- Когда первые несколько байтов каждого блока заняты указателем на следующий блок, чтение полноценного блока требует получения и соединения информации из двух дисковых блоков, из-за чего возникают дополнительные издержки при копировании.

#### Размещение с помощью связанного списка, использующего таблицу в памяти.

Недостатки размещения с помощью связанных списков могут быть устранены за счет изъятия слова указателя из каждого дискового блока и помещения его в таблицу в памяти.

#### *Преимущества:*

- При использовании такой организации для данных доступен весь блок.
- Намного упрощается произвольный доступ. Хотя для поиска заданного смещения в файле по-прежнему нужно идти по цепочке, эта цепочка целиком находится в памяти, поэтому проход по ней может осуществляться без обращений к диску.

#### *Недостатки:*

- Для работы вся таблица должна постоянно находиться в памяти.

## **I-узлы. Индексное размещение.**

методов отслеживания принадлежности конкретного блока конкретному файлу является связь с каждым файлом структуры данных, называемой i-узлом (index-node — индекс-узел), содержащей атрибуты файла и дисковые адреса его блоков.

При индексированном размещении блоки файла могут быть расположены как угодно разрозненно, но индексный блок содержит все ссылки на них.

### *Преимущества:*

- Большим преимуществом этой схемы перед связанными списками, использующими таблицу в памяти, является то, что i-узел должен быть в памяти только в том случае, когда открыт соответствующий файл.

### *Недостатки:*

- на индексный блок ложится критическая нагрузка: если его целостность будет нарушена, файл восстановлению не подлежит.

## **Методы доступа к файлам.**

Файловая система может использовать два способа доступа к логическим записям: читать или записывать логические записи последовательно (последовательный доступ) или позиционировать файл на запись с указанным номером (прямой доступ).

### **Последовательный доступ.**

Процесс может читать все байты или записи файла только по порядку, с самого начала.

### **Произвольный доступ.**

Возможно считывать байты или записи файла вне порядка их размещения или получать доступ к записям по адресу(ключу).

## 19. Методы учета занятого/свободного пространства (ФС)

### Список адресов свободных блоков

Поскольку число свободных индексных дескрипторов и блоков хранения данных может быть значительным, хранение двух последних списков целиком в суперблоке непрактично. Например, для индексных дескрипторов хранится только часть списка. Когда число свободных дескрипторов в этом списке приближается к 0, ядро просматривает массив `ilist` и вновь формирует список свободных дескрипторов `inode`. Для этого ядро анализирует поле `di_mode` индексного дескриптора, которое равно 0 у свободных `inode`.

К сожалению, такой подход неприменим в отношении свободных блоков хранения данных, поскольку по содержимому блока нельзя определить, свободен он или нет. Поэтому необходимо хранить список адресов свободных блоков целиком. Список адресов свободных блоков может занимать несколько блоков хранения данных, но суперблок содержит только один блок этого списка. Первый элемент этого блока указывает на блок, хранящий продолжение списка и так далее. Выделение свободных блоков для размещения файла производится с конца списка суперблока. Когда в списке остается единственный элемент, ядро интерпретирует его как указатель на блок, содержащий продолжение списка. В этом случае содержимое этого блока считывается в суперблок, а сам блок освобождается. Такой подход позволяет использовать под списки дисковое пространство, пропорциональное свободному месту в файловой системе. Другими словами, когда свободного места практически не остается, список адресов свободных блоков целиком помещается в суперблоке.

### Битовая карта

Информация о свободном пространстве в группе хранится не в виде списка свободных блоков, а в виде битовой карты блоков. Карта блоков, связанная с определенной группой цилиндров, описывает свободное пространство во фрагментах, для определения того, свободен данный блок или нет, а ядро анализирует биты фрагментов, составляющих блок.

## **20. Взаимосвязь методов размещения, доступа и учета свободного пространства (ФС)**

Методы размещения, доступа и учета свободного пространства (ФС) взаимосвязаны и влияют друг на друга.

Метод размещения определяет, как файлы будут сохраняться на ФС и каким образом свободное пространство на диске будет управляться. Например, в некоторых методах файлы сохраняются последовательно, а в других - случайным образом. Этот метод также должен учитывать возможность дефрагментации ФС и эффективности использования свободного пространства.

Метод доступа определяет, как пользователи и программы могут получить доступ к файлам на ФС. Он включает в себя права доступа и механизмы защиты файлов, чтобы предотвратить несанкционированный доступ к конфиденциальной информации.

Метод учета свободного пространства определяет, как информация о свободном пространстве сохраняется и управляется. Он включает в себя механизмы мониторинга использования дискового пространства и оповещение пользователей при достижении критического уровня свободного пространства.

Все эти методы должны быть тщательно реализованы и интегрированы друг с другом, чтобы ФС работала эффективно и безопасно.

## 21. Жизненный цикл и состояния процесса

**Процесс** – программа в стадии выполнения. Это динамический объект; возникает в ОС после того, как пользователь/ОС “запускает” программу на выполнение; новая самостоятельная единица вычислительной работы.

процесс рассматривается операционной системой как заявка на потребление всех видов ресурсов, кроме процессорного времени. Ресурс времени распределяется ОС между более мелкими единицами работы – потоками, позволяя распараллелить вычисления в рамках одного процесса.

Жизненный цикл содержит: состояния и переходы. Состояния бывают активными и пассивными (по отношению к возможности использования процессора)

### Типы состояний:

- **выполнение** - активное состояние процесса
- **ожидание** - пассивное ожидает осуществления некоторого события, характерно наличие очередей процессов по ресурсам в соответствии с приоритетами доступа к каждому из этих ресурсов
- **готовность** - пассивное (процесс заблокирован в связи с внешними обстоятельствами); имеет все требуемые ресурсы и готов выполняться, но процессор занят другим процессом;

### Упрощенная последовательность перехода:

- 1) активация процесса -> выполнение
- 2) выполнение (если процесс освободил процессор, т.е. был завершен)-> ожидание
- 3) выполнение (если процесс был насильно “вытеснен”, т.е. исчерпал квант процессорного времени) -> готовность
- 4) ожидание (если произошло ожидаемое событие) -> готовность

### Обобщенное описание состояний:

- 1) **Рождение процесса:** Первый процесс в системе рождается через `init` — порождается непосредственно ядром. Все остальные процессы появляются путём дублирования текущего процесса с помощью системного вызова `fork(2)`.
- 2) **Готов:** Сразу после выполнения `fork(2)` переходит в состояние «готов». Фактически, процесс стоит в очереди и ждет, когда планировщик (`scheduler`) в ядре даст процессу выполняться на процессоре.
- 3) **Выполняется:** Как только планировщик поставил процесс на выполнение, началось состояние «выполняется».
- 4) **Ожидает:** Некоторые системные вызовы могут выполняться долго, например, ввод-вывод. В таких случаях процесс переходит в состояние «ожидает».
- 5) **Остановлен:** в любой момент можно приостановить выполнение процесса, отправив ему сигнал `SIGSTOP`. Процесс перейдет в состояние «остановлен» и будет находиться там до тех пор, пока ему не придёт сигнал продолжать работу (`SIGCONT`) или умереть (`SIGKILL`).
- 6) **Завершение:** Ни одна программа не умеет завершаться сама. Они могут лишь попросить систему об этом с помощью системного вызова `exit` или быть завершёнными системой из-за ошибки.
- 7) **Зомби:** Сразу после того, как процесс завершился (не важно, корректно или нет), ядро записывает информацию о том, как завершился процесс и переводит его в

состояние «зомби». Иными словами, зомби — это завершившийся процесс, но память о нём всё ещё хранится в ядре.

#### Подробный перечень состояний (в ОС Unix):

- 1) Процесс выполняется в пользовательском режиме, или режиме задачи.
- 2) Процесс выполняется в привилегированном режиме, или режиме ядра.
- 3) Процесс не выполняется, но готов к запуску под управлением ядра.
- 4) Процесс приостановлен и находится в оперативной памяти.
- 5) Процесс готов к запуску, но программа подкачки (нулевой процесс) должна еще загрузить процесс в оперативную память, прежде чем он будет запущен под управлением ядра.
- 6) Процесс приостановлен, и программа подкачки выгрузила его во внешнюю память, чтобы в оперативной памяти освободить место для других процессов.
- 7) Процесс возвращен из привилегированного режима (режима ядра) в непривилегированный (режим задачи), ядро резервирует его и переключает контекст на другой процесс.
- 8) Процесс вновь создан и находится в переходном состоянии; процесс существует, но не готов к выполнению, хотя и не приостановлен.
- 9) Процесс вызывает системную функцию exit() и прекращает существование. Однако после него осталась запись, содержащая код выхода, и некоторая хронометрическая статистика, собираемая родительским процессом.



## 22. Контекст и дескриптор процесса на примере Unix

**Контекст процесса** - информация, описывающая состояние процесса и его операционной среды. Эта информация сохраняется, когда выполнение процесса приостанавливается, и восстанавливается, когда планировщик предоставляет процессу вычислительные ресурсы.

**Он объединяет:**

1) **Пользовательский контекст** - команды и данные процесса, стек задачи, содержимое совместно используемого пространства памяти в виртуальных адресах процесса.

2) **Регистровый контекст** - счетчик команд, регистр состояния процесса, указатель вершины стека, регистры общего назначения.

3) **Системный контекст** - запись в таблице процессов, часть адресного пространства задачи, стек ядра, записи частной таблицы областей процесса, динамическая часть системного контекста процесса.

**Контекст содержит две части:**

1) **Статическая** - программы процесса (машинные инструкции), данные, стек, разделяемая память, записи таблицы процессов и др.

2) **Динамическая** - включает элементы, хранящие регистровый контекст предыдущего уровня и стек ядра текущего уровня.

**Дескриптор процесса** содержит такую информацию о процессе, которая необходима ядру в течение всего жизненного цикла процесса, независимо от того, находится ли он в активном или пассивном состоянии, находится ли образ процесса в оперативной памяти или выгружен на диск.

**Он содержит:**

- идентификатор процесса,
- состояние процесса,
- данные о степени привилегированности и др.

Дескрипторы отдельных процессов объединены в список, образующий таблицу процессов. Память для нее отводится динамически в области ядра. На ее основе ОС осуществляет планирование и синхронизацию процессов.



## 23. Структуры контекстов процессов и нитей

В контексте потока можно выделить часть, общую для всех потоков данного процесса (ссылки на открытые файлы), и часть, относящуюся только к данному потоку (содержимое регистров, счетчик команд, режим процессора). Например, в среде NetWare 4.x различаются три вида контекстов: глобальный контекст (контекст процесса), контекст группы потоков и контекст отдельного потока. Соотношение между данными этих контекстов напоминает соотношение глобальных и локальных переменных в программе, написанной на языке C. Переменные глобального контекста доступны для всех потоков, созданных в рамках одного процесса. Переменные локального контекста доступны только для кодов определенного потока, аналогично локальным переменным функции. В NetWare можно создавать несколько групп потоков внутри одного процесса и эти группы будут иметь свой групповой контекст. Переменные, принадлежащие групповому контексту, доступны всем потокам, входящим в группу, но недоступны остальным потокам.

### Структура процесса

Процесс в ядре представляется просто как структура с множеством полей, например:

- Идентификатор процесса (pid)
- Открытые файловые дескрипторы (fd)
- Обработчики сигналов (signal handler)
- Текущий рабочий каталог (cwd)
- Переменные окружения (environ)
- Код возврата

## **24. Переключение контекста и затраты ресурсов на переключение**

Переключение между процессами, необходимое для справедливого распределения вычислительного ресурса, по существу выражается в переключении контекста, когда контекст выполнявшегося процесса запоминается и восстанавливается контекст процесса, выбранного планировщиком. Переключение контекста является достаточно ресурсоемкой операцией. Помимо сохранения состояния регистров процесса, ядро вынуждено выполнить множество других действий.

Существуют четыре ситуации, при которых производится переключение контекста:

- Текущий процесс переходит в состояние сна, ожидая недоступного ресурса

- Текущий процесс завершает свое выполнение

- После пересчета приоритетов в очереди на выполнение находится более высокоприоритетный процесс

- Происходит пробуждение более высокоприоритетного процесса

## 25. Методы и алгоритмы планирования

В ОС общего назначения или иных ОС с высокой степенью универсальности по применению обычно предполагается, что входной поток задач случайный. При такой степени неопределенности самое разумное, что можно сделать для оптимизации вычислительного процесса в системе в целом – это минимизировать затраты ресурсов на само планирование, что делается за счет выбора самых простейших алгоритмов планирования и сокращения их количества в ОС. Это отражено в стандарте POSIX в виде рекомендуемых алгоритмов: круговое планирование и FIFO.

Основные типы алгоритмов представлены в таблице ниже.

Дисциплина планирования	Порядок обслуживания заданий
<b>RMS rate monotonic scheduling</b>	Задания выполняются в порядке увеличения значений периода поступления <i>(требуют, чтобы прикладные задачи были периодическими, с постоянством интервалов времени использования центрального процессора)</i>
<b>DMS deadline monotonic scheduling</b>	Задания выполняются в порядке увеличения значений относительного срока выполнения
<b>EDF earliest deadline first</b>	Задания выполняются в порядке увеличения значений абсолютного срока завершения <i>(запускается процесс, у которого раньше всех наступит крайний срок. Как только будет готов новый процесс, система проверяет, не наступает ли его крайний срок раньше, чем у процесса, работающего в данный момент)</i>
<b>LLF least laxity first</b>	Задания выполняются в порядке увеличения запаса времени на выполнение
<b>LWR least work remaining</b>	Задания выполняются в порядке увеличения оставшейся длительности обслуживания
<b>LIFO last in – first out</b>	Процессор предоставляется заданию с наиболее поздним моментом времени регистрации
<b>SPT shortest processing time</b>	Задания выполняются в порядке увеличения значений объема требуемых вычислений

Сегодня в Unix подобных ОС, в частности в Linux, и других ОС, следующих стандарту POSIX, поддерживаются три базовые политики планирования: SCHED\_FIFO, SCHED\_RR, и SCHED\_OTHER: одна для обычных процессов и две для процессов «реального» времени. Их реализация обеспечивается ядром, а точнее, планировщиком. Каждому процессу присваивается **статический приоритет** sched\_priority, который можно изменить только при помощи системных вызовов. Ядро хранит в памяти списки всех работающих процессов для каждого возможного значения sched\_priority. Для того, чтобы определить, какой процесс будет выполняться следующим, планировщик ищет непустой список (очередь) с наибольшим статическим приоритетом и запускает первый процесс из этого списка.

**1) SCHED\_FIFO: планировщик FIFO (First In-First Out).** Можно использовать только со значениями статического приоритета, большими нуля. Это означает, что если процесс готов к работе, то он сразу запустится, а все обычные процессы с алгоритмом SCHED\_OTHER будут приостановлены.

**2) SCHED\_RR: циклический алгоритм планирования.** Все, относящееся к алгоритму SCHED\_FIFO, справедливо и для SCHED\_RR за исключением того, что каждому процессу разрешено работать непрерывно не дольше некоторого времени, называемого квантом.

**3) SCHED\_OTHER: стандартный алгоритм планировщика с разделением времени.** Можно использовать только со значениями статического приоритета, равными нулю. Предназначенный для процессов, не требующих специальных механизмов реального времени со статическими приоритетами. Порядок предоставления процессорного времени процессам со статическим приоритетом, равным нулю, основывается на **динамических приоритетах**, существующих только внутри этого списка. Динамический приоритет основан на уровне nice и увеличивается с каждым квантом времени, при котором процесс был готов к работе, но ему было отказано в этом планировщиком.

## 26. Инверсия приоритетов

В Linux инверсия приоритетов может возникать, когда задача с низким приоритетом блокирует ресурсы, необходимые для выполнения задачи с более высоким приоритетом. Это может произойти, например, когда задача с низким приоритетом запрашивает доступ к файлу или использует сетевые ресурсы, которые нужны для выполнения задачи с более высоким приоритетом.

Для предотвращения инверсии приоритетов в Linux используются различные механизмы. Например, планировщик задач в Linux использует приоритеты процессов и потоков для управления выполнением задач. Задачи с более высоким приоритетом получают больше времени на выполнение, чем задачи с более низким приоритетом.

Кроме того, в Linux используются механизмы управления блокировками, которые позволяют избежать блокировки выполнения задач с более высоким приоритетом.

Мьютексы реального времени обеспечивают наследование приоритетов (priority inheritance, PI), что является одним из нескольких (немногих) известных способов, препятствующих возникновению инверсии приоритетов (priority inversion). Если RT мьютекс захвачен процессом А, и его пытается захватить процесс В (более высокого приоритета), то:

- процесс В блокируется и помещается в очередь ожидающих освобождения процессов `wait_list` (в описании структуры `rt_mutex`);
- при необходимости, этот список ожидающих процессов переупорядочивается в порядке приоритетов ожидающих процессов;
- приоритет владельца мьютекса (текущего выполняющегося процесса) В повышается до приоритета ожидающего процесса А (максимального приоритета из ожидающих в очереди процессов);
- это и обеспечивает избежание потенциальной инверсии приоритетов.

## 27. Методы IPC в Unix-подобных ОС: состав и общая характеристика каждого типа IPC (включая сетевые)

### 1) Сигналы.

Сигналы в системе UNIX используются для того, чтобы: сообщить процессу о том, что возникло асинхронное событие; или необходимо обработать исключительное состояние.

Сигнал позволяет передать уведомление о некотором произошедшем событии между процессами или между ядром системы и процессами. Это означает, что посредством сигналов можно выполнять две основные функции IPC: передачу информации и синхронизацию процессов или потоков.

Для отправки и доставки сигнала требуется системный вызов. Для доставки – прерывание и его обработка.

В Unix системах можно выделить три типа сигналов: надежные, ненадежные и сигналы реального времени (Real-time signals).

Надежные сигналы (reliable signals) гарантируют, что сигнал будет доставлен и обработан только один раз.

Ненадежные сигналы (unreliable signals) могут быть доставлены несколько раз и в случайном порядке, что может привести к проблемам обработки.

Сигналы реального времени (Real-time signals) — это надежный, асинхронный механизм межпроцессного взаимодействия, который позволяет передавать пользовательские данные с сигналом. В отличие от обычных сигналов, сигналы реального времени гарантируют доставку сигнала в порядке его отправки и могут быть использованы для передачи более сложной информации, например, вектора событий.

### 2) Каналы

Различают два типа каналов анонимные (неименованные) и именованные. Они по-разному реализованы, но доступ к ним организуется одинаково с помощью обычных функций read и write. Одним из свойств программных каналов и FIFO является то, что данные по ним передаются в виде потоков байтов.

Неименованные каналы: являются однонаправленными и имеют ограниченную пропускную способность. Они создаются с помощью системного вызова pipe() и могут быть использованы только между процессами, которые являются родственными.

Именованные каналы: (FIFO) также являются однонаправленными, но имеют более широкий спектр применения. Они создаются с помощью системного вызова mkfifo() и могут быть использованы между процессами, которые не являются родственными. Именованный канал является файлом, который хранится в файловой системе.

На неименованные каналы и каналы FIFO системой накладываются всего два ограничения: OPEN\_MAX — максимальное количество дескрипторов, которые могут

быть одновременно открыты некоторым процессом, PIPE\_BUF — максимальное количество данных, для которого гарантируется атомарность операции записи

### 3) Очередь сообщений

Очередь сообщений находится в адресном пространстве ядра и имеет ограниченный размер. В отличие от каналов, которые обладают теми же самыми свойствами, очереди сообщений сохраняют границы сообщений. Это значит, что ядро ОС гарантирует, что сообщение, поставленное в очередь, не смешается с предыдущим или следующим сообщением при чтении из очереди. Кроме того, с каждым сообщением связывается его тип. Процесс, читающий очередь сообщений, может отбирать только сообщения заданного типа или все сообщения кроме сообщений заданного типа.

Каждое сообщение представляет собой запись, очереди сообщений автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP.

### 4) Семафоры и разделяемая память

Семафоры - это механизм синхронизации в Unix-подобных операционных системах, который позволяет контролировать доступ к разделяемым ресурсам в многопроцессном окружении.

Семафоры могут работать в разных режимах:

- **Бинарные** (0 или 1), рассматриваются как флаги при создании семафоров и обычно используются для разделения доступа к ресурсам.
- **Счетные**, имеют числовое значение и используются для контроля количества доступов к ресурсам.

Функция `semget()` создает массив семафоров или возвращает идентификатор существующего массива. Функция `semop()` позволяет изменять значения семафоров в массиве, путем блокировки, разблокировки или ожидания. Функция `semctl()` используется для управления семафорами в массиве, например, для удаления массива семафоров.

Применение семафоров может свести к минимуму конфликты и задержки при доступе к разделяемым ресурсам, что делает их полезными в разработке многопоточных и распределенных систем.

**Разделяемая память (Shared memory)** - это механизм в Unix-подобных операционных системах, который позволяет нескольким процессам обмениваться данными через общую область памяти. Этот механизм позволяет исключить затраты на передачу больших объемов данных между процессами и ускорить обмен данными.

**Процедура создания общей памяти включает в себя следующие шаги:**

1. Создание сегмента общей памяти с помощью системного вызова `shmget()`.
2. Присоединение сегмента общей памяти к адресному пространству процесса с помощью системного вызова `shmat()`.
3. Работа с общей памятью, как с обычной областью памяти.

4. Отсоединение сегмента общей памяти от процесса с помощью системного вызова `shmdt()`.

5. Удаление сегмента общей памяти с помощью системного вызова `shmctl()`.

Разделяемая память может использоваться для обмена большими объемами данных между процессами, обмена данными между процессами в режиме реального времени, реализации межпроцессного взаимодействия.

## 6) Сокеты

Программисты в Unix-подобных операционных системах используют сокеты (sockets) для реализации механизмов межпроцессного взаимодействия. Сокеты - это механизм, который позволяет обеспечивать связь между двумя процессами через сеть или локальный компьютер.

Сокеты в Unix-подобных операционных системах описываются файлом дескриптора (file descriptor). Как правило, созданием, открытием, закрытием, отправкой и получением сообщений сокетов занимается прикладной уровень.



## 28. Обработка событий

**Обработка событий в операционной системе** - это процесс обработки событий, которые происходят в операционной системе. События могут быть вызваны пользователем или программой. Обработка событий может включать в себя запись информации о событии в журнале событий, запуск другой программы или выполнение других действий.

В Linux обработка событий осуществляется через систему файловых дескрипторов

Обработка событий в операционной системе Linux осуществляется с использованием механизмов ядра и различных служб операционной системы. **Вот общая схема работы обработки событий в Linux:**

1) **Ядро операционной системы:** Ядро Linux отвечает за управление всеми ресурсами компьютера, включая процессор, память, диски и устройства ввода-вывода. Ядро также отвечает за планирование выполнения процессов и управление событиями в системе

2) **Прерывания:** Прерывания – это механизм, позволяющий устройствам прерывать обычный ход выполнения программы и привлекать внимание ядра. Когда устройство генерирует прерывание, ядро обрабатывает его и вызывает соответствующий обработчик прерывания, который выполняет необходимые действия

3) **Обработчики прерываний:** часть ядра Linux, которая вызывается в ответ на прерывание устройства. Обработчики прерываний выполняются с высоким приоритетом и обрабатывают прерывание, выполняя необходимые действия, например, чтение данных с устройства или запись данных на диск

4) **Планировщик:** Планировщик ядра отвечает за управление выделением процессорного времени между различными процессами. Когда процесс завершает выполнение или происходит событие, требующее переключения контекста, планировщик выбирает следующий процесс для выполнения и переключает контекст выполнения на него

5) **Сигналы:** механизм взаимодействия между процессами и ядром. Процессы или ядро могут отправлять сигналы другим процессам для уведомления о различных событиях. Процессы могут устанавливать обработчики сигналов для обработки событий, связанных с сигналами

6) **Сокеты и файловые дескрипторы:** Linux предоставляет механизмы сокетов и файловых дескрипторов для обработки событий ввода-вывода. Процессы могут ожидать событий ввода-вывода из сокетов или файловых дескрипторов и блокироваться до наступления события. Когда событие происходит, процесс разблокируется и может обработать событие

## 29. Синхронизация при разделении ресурсов, средства синхронизации

### Цели и средства синхронизации

Существует достаточно обширный класс средств операционной системы, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков возникает только в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Любое взаимодействие процессов или потоков связано с их синхронизацией, которая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. Синхронизация лежит в основе любого взаимодействия потоков, связано ли это взаимодействие с разделением ресурсов или с обменом данными.

### Критическая секция

*Критическая секция* – это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено.

Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток. При этом неважно, находится этот поток в активном или в приостановленном состоянии. Этот прием называют взаимным исключением.

### Блокирующие переменные

Для синхронизации потоков одного процесса прикладной программист может использовать *глобальные блокирующие переменные*. С этими переменными, к которым все потоки процесса имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС.

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рисунке 3.11 показан фрагмент алгоритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную F(D).

Если все потоки написаны с учетом вышеописанных соглашений, то взаимное исключение гарантируется. При этом потоки могут быть прерваны операционной системой в любой момент и в любом месте, в том числе в критической секции.

Нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной.

## Семафоры

Обобщением блокирующих переменных являются так называемые семафоры Дийкстры. Вместо двоичных переменных Дийкстра (Dijkstra) предложил использовать переменные, которые могут принимать целые неотрицательные значения. Такие переменные, используемые для синхронизации вычислительных процессов, получили название семафоров.

Для работы с семафорами вводятся два примитива, традиционно обозначаемых  $P$  и  $V$ . Пусть переменная  $S$  представляет собой семафор. Тогда действия  $V(S)$  и  $P(S)$  определяются следующим образом.

- $V(S)$ : переменная  $S$  увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной  $S$  нет доступа другим потокам во время выполнения этой операции.
- $P(S)$ : уменьшение  $S$  на 1, если это возможно. Если  $S = 0$  и невозможно уменьшить  $S$ , оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию  $P$ , ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Никакие прерывания во время выполнения примитивов  $V$  и  $P$  недопустимы.

## Синхронизирующие объекты ОС

Рассмотренные выше механизмы синхронизации, основанные на использовании глобальных переменных процесса, обладают существенным недостатком — они не подходят для синхронизации потоков разных процессов.

Примерами синхронизирующих объектов ОС являются системные семафоры, мьютексы, события, таймеры и другие — их набор зависит от конкретной ОС, которая создает эти объекты по запросам процессов.

Объект-файл, переход которого в сигнальное состояние соответствует завершению операции ввода-вывода с этим файлом, используется в тех случаях, когда поток, инициировавший эту операцию, решает дождаться ее завершения, прежде чем продолжить свои вычисления.

Мьютекс, как и семафор, обычно используется для управления доступом к данным.

В отличие от объектов-потоков, объектов-процессов и объектов-файлов, которые при переходе в сигнальное состояние переводят в состояние готовности все потоки, ожидающие этого события, объект - мьютекс «освобождает» из очереди ожидающих только один поток.

Объект-событие (в данном случае слово «событие» используется в узком смысле, как обозначение конкретного вида объектов синхронизации) обычно используется не для доступа к данным, а для того, чтобы оповестить другие потоки о том, что некоторые действия завершены. Пусть, например, в некотором приложении работа организована

таким образом, что один поток читает данные из файла в буфер памяти, а другие потоки обрабатывают эти данные, затем первый поток считывает новую порцию данных, а другие потоки снова ее обрабатывают и так далее. В начале работы первый поток устанавливает объект-событие в несигнальное состояние. Все остальные потоки выполнили вызов Wait(X), где X — указатель события, и находятся в приостановленном состоянии, ожидая наступления этого события. Как только буфер заполняется, первый поток сообщает об этом операционной системе, выполняя вызов Set(X). Операционная система просматривает очередь ожидающих потоков и активизирует все потоки, которые ждут этого события.

## Сигналы

*Сигнал* дает возможность задаче реагировать на событие, источником которого может быть операционная система или другая задача. Сигналы вызывают прерывание задачи и выполнение заранее предусмотренных действий.

Сигналы могут вырабатываться:

1) *синхронно*, то есть как результат работы самого процесса. Чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например, деление на нуль, ошибка адресации, нарушение защиты памяти и т. д.;

2) *асинхронно*, то есть направлены процессу другим процессом. Например, сигнал с терминала. Во многих ОС предусматривается оперативное снятие процесса с выполнения. Для этого пользователь может нажать некоторую комбинацию клавиш (Ctrl+C, Ctrl+Break), в результате чего ОС вырабатывает сигнал и направляет его активному процессу. Сигнал может поступить в любой момент выполнения процесса, требуя от процесса немедленного завершения работы. В данном случае реакцией на сигнал является безусловное завершение процесса.

В системе может быть определен набор сигналов.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, которые могут получить данные об идентификаторах друг друга.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, блокирующие переменные, семафоры, сигналы и другие аналогичные средства, основанные на разделяемой памяти, оказываются непригодными. В таких системах синхронизация может быть реализована только посредством *обмена сообщениями.*

### 30. Типы памяти. Виды организации виртуальной памяти

#### По назначению:

- **Буферная память** — память, предназначенная для временного хранения данных при обмене ими между различными устройствами или программами.
- **Временная (промежуточная) память** — память для хранения промежуточных результатов обработки.
- **Кеш-память** — часть архитектуры устройства или программного обеспечения, осуществляющая хранение часто используемых данных для предоставления их в более быстрый доступ, нежели кешируемая память.
- **Корректирующая память** — часть памяти ЭВМ, предназначенная для хранения адресов неисправных ячеек основной памяти. Также используются термины relocation table и remap table.
- **Управляющая память** — память, содержащая управляющие программы или микропрограммы. Обычно реализуется в виде ПЗУ.
- **Разделяемая память или память коллективного доступа** — память, доступная одновременно нескольким пользователям, процессам или процессорам.

#### По организация адресного пространства:

- **Реальная или физическая память** — память, способ адресации которой соответствует физическому расположению её данных;
- **Виртуальная память** — память, способ адресации которой не отражает физического расположения её данных;
- **Оверлейная память** — память, в которой присутствует несколько областей с одинаковыми адресами, из которых в каждый момент доступна только одна.

#### По удалённость и доступность для процессора:

- **Первичная память** (сверхоперативная, СОЗУ) — доступна процессору без какого-либо обращения к внешним устройствам. Отличается крайне малым временем доступа и недоступна для явного использования в программе.
- **Вторичная память** — доступна процессору путём прямой адресации через шину адреса (адресуемая память). Таким образом доступна оперативная память и порты ввода-вывода (специальные адреса, через обращение к которым реализовано взаимодействие с прочей аппаратурой).
- **Третичная память** — доступна только путём нетривиальной последовательности действий. Сюда входят все виды внешней памяти — доступной через устройства ввода-вывода. Взаимодействие с третичной памятью ведётся по определённым правилам (протоколам) и требует присутствия в памяти соответствующих программ.

#### По доступности техническими средствами:

- **Непосредственно управляемая** (оперативно доступная) память — память, непосредственно доступная в данный момент.

- **Автономная память** — память, доступ к которой требует внешних действий — например, вставку оператором архивного носителя с указанным программой идентификатором.

- **Полуавтономная память** — то же, что автономная, но физическое перемещение носителей осуществляется роботом по команде системы, то есть не требует присутствия оператора

#### **Виды организации виртуальной памяти:**

- ✓ Страничная
- ✓ Сегментная

Страничная виртуальная память организует перемещение данных между памятью и диском страницами — частями виртуального адресного пространства, фиксированного и сравнительно небольшого размера.

Сегментная виртуальная память предусматривает перемещение данных сегментами — частями виртуального адресного пространства произвольного размера, полученными с учетом смыслового значения данных. Физический адрес вычисляется по формуле:  $addr = base + offset$ .

Сегментно-страничная виртуальная память использует двухуровневое деление: виртуальное адресное пространство делится на сегменты, а затем сегменты делятся на страницы. Единицей перемещения данных здесь является страница. Этот способ управления памятью объединяет в себе элементы обоих предыдущих подходов.



## **31. Управление памятью, распределенной динамическими разделами**

### **Распределение динамическими разделами**

При распределении динамическими разделами память выделяется по запросам. В запросе содержится объем требуемого участка памяти.

Запросы выдаются программой управления процессами при создании очередного процесса.

Система управления памятью при удовлетворении запроса находит свободный участок требуемого объема и создает раздел процесса. При завершении процесса этот раздел уничтожается.

### **Реализация функций**

При способе управления памятью, распределяемой динамическими разделами, необходимо учитывать свободные и занятые участки, т.е. хранить информацию о базовых адресах и размерах этих участков памяти. Во многих ОС используется способ учета с помощью списка блоков управления памятью.

С этим списком работает программа ядра, занимающаяся выделением памяти. Для освобождения памяти необходимо знать адрес и размер занятого участка. Эта информация хранится в блоке управления процессом. При уничтожении процесса освобождается занятый участок, и эта информация используется для реорганизации списка блоков свободной памяти.

Планирование запросов на выделение памяти осуществляется так же, как и при распределении статическими разделами. Программы планирования процессов или системный загрузчик выдают запрос для очередного созданного или загружаемого процесса. Объем памяти предоставляемый по запросу, должен быть не менее запрашиваемого объема.

### **Алгоритмы программного обеспечения**

При выделении памяти необходимо просмотреть список МСВ и найти подходящий участок. Наиболее популярны следующие алгоритмы просмотра.

#### **Алгоритм "первый подходящий"**

Выбор первого свободного участка, размер которого больше требуемого объема. В этом случае выбранный свободный участок делится, и оставшаяся часть свободного участка включается в список на место выбранного свободного участка. Таким образом, крупные свободные участки сдвигаются в конец списка.

С одной стороны, это положительный момент, так как запросы на малые участки удовлетворяются в одной области памяти, а на большие - в другой. Это уменьшает дробление памяти на небольшие участки, уменьшает фрагментацию.

С другой стороны, увеличивается время поиска свободного участка, так как в начале списка группируются участки малого размера. Кроме того, существует опасность, что оставшаяся свободная память от выбранного участка будет настолько мала, что не удовлетворит никакому запросу.

В этом случае целесообразно установить некоторый порог, и в случае, если размер оставшегося свободного участка меньше этого порогового значения, то следует занять весь выбранный участок. Чтобы участки малого размера не группировались в начале списка, список можно закольцевать, а указатель на начало списка перемещать, т.е. после выделения свободного участка указатель содержит адрес следующего свободного участка. Просмотр при удовлетворении следующего запроса начнется с другого элемента списка.

### **Алгоритм "самый подходящий"**

Выбор свободного участка наименьшего размера, удовлетворяющего запросу.

В этом случае можно использовать такой же алгоритм работы со списком, как и алгоритм "первый подходящий", но список должен быть упорядочен по возрастанию размеров свободных участков. При подобной организации списка первый найденный свободный участок и будет самым подходящим.

#### **Достоинства:**

1) Если существует свободный участок, размер которого в точности соответствует размеру запроса, то этот участок будет выбран.

2) При использовании данного алгоритма свободные области больших размеров остаются нетронутыми.

#### **Недостатки:**

1) Увеличение вероятности получения маленьких остатков и, как следствие, увеличение дробления памяти.

2) При использовании упорядоченного по возрастанию списка необходимо просматривать список для поиска места, куда должен быть вставлен новый образовавшийся свободный блок.

### **Достоинства и недостатки распределения динамическими разделами**

#### **Достоинства:**

1) Возможность организации мультипрограммного режима работы

#### **Недостатки:**

1) Усложнение алгоритмов управления и системных структур данных

2) Память используется неэффективно по причине загрузки редко используемого логического адресного пространства

3) Фрагментация ОП



## 32. Механизмы преобразования виртуального адреса в физический при различных организациях памяти

Механизмы преобразования виртуального адреса в физический адрес при различных организациях памяти могут отличаться, но общий принцип заключается в том, что процессор преобразует виртуальный адрес (вычисленный программой) в физический адрес в памяти. Это необходимо для того, чтобы процессор мог обратиться к нужной ячейке памяти для чтения или записи данных.

Рассмотрим несколько организаций памяти и их механизмы преобразования виртуального адреса.

1. Организация памяти с постоянным размещением (Fixed partition memory organization). В этой организации память разбивается на фиксированные блоки, каждый из которых имеет свой физический адрес. Виртуальный адрес в данном случае состоит из номера раздела (partition number) и смещения (offset). Для преобразования виртуального адреса в физический процессор использует формулу  $\text{физический адрес} = \text{базовый адрес} + \text{смещение}$ . Базовый адрес для каждого раздела можно сохранить в таблице фиксированных разделов (Fixed Partition Table).

2. Организация памяти с динамическим размещением (Dynamic partition memory organization). В этой организации память разбивается на блоки произвольного размера, которые могут быть использованы для размещения программ и данных. Для управления блоками в памяти используется таблица страниц (Page table), которая содержит информацию о каждом блоке в памяти, включая его физический адрес и состояние (занят/свободен). Виртуальный адрес в данном случае состоит из номера страницы (page number) и смещения (offset). Для преобразования виртуального адреса в физический процессор использует таблицу страниц, которая содержит соответствие между виртуальными и физическими адресами блоков в памяти.

3. Организация памяти сегментированной системой (Segmented memory organization). В этой организации память разбивается на разделы (сегменты), которые могут содержать код программы, данные и стек вызовов функций. Каждый сегмент имеет свой физический адрес. Виртуальный адрес в данном случае состоит из номера сегмента и смещения внутри сегмента. Процессор использует таблицу сегментов (Segment table), которая хранит информацию о каждом сегменте, включая его начальный адрес и размер. Для преобразования виртуального адреса в физический, процессор использует таблицу сегментов, чтобы определить физический адрес сегмента, и затем добавляет к нему смещение внутри сегмента.

Каждая из этих организаций памяти имеет свои преимущества и недостатки, и выбор зависит от требований к производительности, экономичности и безопасности.

### 33. Базовые сервисы диспетчера виртуальной памяти

Система виртуальной памяти в Linux поддерживает адресное пространство, видимое каждому процессу: она создает страницы виртуальной памяти по требованию и управляет загрузкой этих страниц с диска или откачкой их обратно на диск, если требуется.

Менеджер виртуальной памяти поддерживает **две точки зрения на адресное пространство каждого процесса:**

- **Логическую** – поддержка команд управления адресным пространством. Адресное пространство рассматривается как совокупность непересекающихся смежных областей.
- **Физическую** – с помощью таблицы страниц для каждого процесса.

**Для управления виртуальной памятью используются:**

- **Файл откочки (backing store)**, описывающий, откуда берутся страницы для заданного региона; регионы обычно поддерживаются либо файлом, либо не поддерживаются вообще (**память, обнуляемая по требованию**)
- **Реакция региона на запись** (совместное использование страниц или копирование при записи - COW).

Ядро создает новое виртуальное адресное пространство:

- Когда процесс запускает новую программу системным вызовом `exec`;
- При создании нового процесса системным вызовом `fork`.

При исполнении новой программы процессу предоставляется новое, пустое адресное пространство; процедуры загрузки программ наполняют это адресное пространство регионами виртуальной памяти.

Создание нового процесса с помощью `fork` включает создание полной копии адресного пространства существующего процесса.

Ядро копирует дескрипторы доступа к виртуальной памяти родительского процесса, затем создает новый набор таблиц страниц для дочернего процесса.

Таблицы страниц процесса-родителя копируются непосредственно в таблицы страниц дочернего, причем счетчик ссылок на каждую страницу увеличивается.

После исполнения `fork` родительский и дочерний процесс используют одни и те же физические страницы в своих виртуальных адресных пространствах.

Система управления страницами откачивает страницы физической памяти на диск, если они требуются для какой-либо другой цели.

**Система управления страницами делится на две части:**

- Алгоритм откочки, который определяет, какие страницы и когда откачать на диск;

- Механизм подкачки фактически выполняет передачу и подкачивает данные обратно в физическую память, если требуется.

Ядро Linux резервирует постоянный, зависящий от архитектуры регион виртуального адресного пространства каждого процесса для его собственного внутреннего использования.

Эта область виртуальной памяти ядра содержит два региона:

- Статическую область, содержащую ссылки из таблицы страниц на каждую доступную физическую страницу памяти в системе, так что используется простая трансляция физического адреса в виртуальный при исполнении кода ядра.
- Остаток зарезервированной части не используется ни для какой другой цели; его элементы таблицы страниц могут быть модифицированы и указывать на любые страницы в памяти.

### 34. Принцип кэширования данных. Стратегия подкачки страниц. Свопинг.

**Кэш-память** - это способ совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который за счет динамического копирования в «быстрое» ЗУ наиболее часто используемой информации из «медленного» ЗУ позволяет, с одной стороны, уменьшить среднее время доступа к данным, а с другой стороны, экономить более дорогую быстродействующую память.

В системах, оснащенных кэш-памятью, каждый запрос к оперативной памяти выполняется в соответствии со следующим алгоритмом:

- Просматривается содержимое кэш-памяти с целью определения, не находятся ли нужные данные в кэш-памяти; кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому - значению поля "адрес в оперативной памяти", взятому из запроса.
- Если данные обнаруживаются в кэш-памяти, то они считываются из нее, и результат передается в процессор.
- Если нужных данных нет, то они вместе со своим адресом копируются из оперативной памяти в кэш-память, и результат выполнения запроса передается в процессор. При копировании данных может оказаться, что в кэш-памяти нет свободного места, тогда выбираются данные, к которым в последний период было меньше всего обращений, для вытеснения из кэш-памяти. Если вытесняемые данные были модифицированы за время нахождения в кэш-памяти, то они переписываются в оперативную память. Если же эти данные не были модифицированы, то их место в кэш-памяти объявляется свободным.

На практике в кэш-память считывается не один элемент данных, к которому произошло обращение, а целый блок данных, это увеличивает вероятность так называемого "попадания в кэш", то есть нахождения нужных данных в кэш-памяти.

В реальных системах вероятность попадания в кэш составляет примерно 0,9. Высокое значение вероятности нахождения данных в кэш-памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

- *Пространственная локальность*. Если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.
- *Временная локальность*. Если произошло обращение по некоторому адресу, то следующее обращение по этому же адресу с большой вероятностью произойдет в ближайшее время.

**Подкачка страниц** - один из механизмов виртуальной памяти, при котором отдельные фрагменты памяти (обычно неактивные) перемещаются из ОЗУ во вторичное хранилище (жёсткий диск или другой внешний накопитель, такой как флеш-память), освобождая ОЗУ для загрузки других активных фрагментов памяти. Такими фрагментами в современных ЭВМ являются страницы памяти.

Описание алгоритма **свопинга** можно разбить на три части: управление пространством на устройстве выгрузки, выгрузка процессов из основной памяти и подкачка процессов в основную память.

Свопинг представляет собой частный случай виртуальной памяти и, следовательно, более простой в реализации способ совместного использования оперативной памяти и диска. Однако подкачке свойственна избыточность: когда ОС решает активизировать процесс, для его выполнения, как правило, не требуется загружать в оперативную память все его сегменты полностью — достаточно загрузить небольшую часть кодового сегмента с подлежащей выполнению инструкцией и частью сегментов Данных, с которыми работает эта инструкция, а также отвести место под сегмент стека. Аналогично при освобождении памяти для загрузки нового процесса очень часто вовсе не требуется выгружать другой процесс на диск целиком, достаточно вытеснить на диск только часть его образа. Перемещение избыточной информации замедляет работу системы, а также приводит к неэффективному использованию памяти. Кроме того, системы, поддерживающие свопинг, имеют еще один очень существенный недостаток: они не способны загрузить для выполнения процесс, виртуальное адресное пространство которого превышает имеющуюся в наличии свободную память.

### 35. Совместное использование памяти процессами.

В большинстве случаев отдельные вычислительные процессы не общаются друг с другом, выполняя самостоятельную и никак не связанную работу. Но существуют задачи, для решения которых создается не один процесс, а несколько, которые, работая совместно, выполняют общую работу. Такие процессы называются **кооперативными**.

Для взаимодействия процессов, в частности кооперативных, сложился ряд средств, часть которых – семафоры уже рассматривались, а некоторые будут изучаться в следующих главах. При всем их многообразии следует вспомнить, что основной информационный ресурс, используемый программами – оперативная память – это самый универсальный и быстрый ресурс компьютера. Поэтому не удивительно, что в состав всех современных ОС включены средства использования памяти для взаимодействия независимых процессов. Основой этих средств является **разделяемая память**, сами средства представляют набор системных функций использования этой памяти, а на более детальном уровне – и информационные средства описания этой памяти.

Изучение средств совместного использования памяти начнем с ОС Unix. Здесь разделяемая память находится под непосредственным управлением ядра, которое содержит таблицу описания областей разделяемой памяти. Каждая из областей обозначается в этой таблице целочисленным идентификатором (а не текстовым именем, как в других ОС). Кроме того, каждая такая область описывается в этой таблице атрибутами доступа и размером. Области разделяемой памяти относятся к адресному пространству ядра ОС.

Доступ к разделяемой памяти со стороны процесса осуществляется в два этапа. На первом из них получается хэндл области памяти, причем на этом этапе либо открывается доступ к уже имеющейся в ОС области памяти, либо такая область создается операционной системой. На втором этапе процесс подключается к разделяемой области, используя ранее полученный хэндл.

На этапе подключения происходит подсоединение указанной области памяти, находящейся в ведении ОС, к виртуальному адресному пространству процесса, запросившего такое подключение. Результатом этой операции является базовый виртуальный адрес, начиная с которого в текущем процессе можно обращаться к разделяемой памяти.

### **36. Защита памяти. Средства преобразования. Распределение памяти между системными и прикладными задачами**

**Защита памяти** — это способ управления правами доступа к отдельным регионам памяти. Используется большинством многозадачных операционных систем. Основной целью защиты памяти является запрет доступа процессу к той памяти, которая не выделена для этого процесса. Такие запреты повышают надежность работы как программ, так и операционных систем, так как ошибка в одной программе не может повлиять непосредственно на память других приложений.

**Преобразование** памяти в операционных системах обычно происходит с помощью специальных механизмов, таких как страницы и сегменты памяти.

**Страницы памяти** - это небольшие блоки памяти фиксированного размера, которые используются для управления виртуальной памятью. Система управления памятью разделяет физическую память на страницы определенного размера и создает соответствующие виртуальные адреса для каждой страницы. Это позволяет приложениям использовать большой объем памяти, намного превышающий фактическую физическую память компьютера.

**Сегменты памяти** - это логические блоки памяти, которые используются для управления кодом, данными и стеком приложения. Каждый сегмент имеет свой собственный начальный адрес и размер, который определяется во время компиляции или загрузки программы.

Операционные системы могут использовать различные стратегии для управления памятью, такие как пейджинг, виртуальная память и фрагментация памяти. Все эти механизмы позволяют эффективно использовать ресурсы компьютера и максимально оптимизировать работу приложений.

В операционных системах используется механизм виртуальной памяти для **распределения** физической памяти между системными и прикладными задачами. Каждой задаче выделяется свой адресное пространство, которое может превышать объем физической памяти в системе.

Операционная система использует страничное преобразование для трансляции виртуальных адресов, используемых задачами, в физические адреса в памяти. Каждая страница памяти может иметь свой признак доступа и прав на чтение/запись/исполнение. Операционная система может также использовать различные алгоритмы для определения, какие страницы должны быть сохранены в физической памяти, а какие могут быть выгружены на жесткий диск.

В целом, операционная система старается как можно более эффективно использовать имеющуюся память в системе, давая приоритет системным задачам, которые обычно потребляют меньше памяти по сравнению с прикладными задачами.



### **37. Состав, функции, структура системы ввода/вывода. BIOS, EFI/UEFI.**

**базовая система ввода-вывода (Basic Input Output System – BIOS)** является, с одной стороны, составной частью аппаратных средств, с другой – одним из программных модулей ОС. Возникновение данного названия связано с тем, что BIOS включает в себя набор программ ввода-вывода. С помощью этих программ ОС и прикладные программы могут взаимодействовать как с различными устройствами самого компьютера, так и с периферийными устройствами.

Как составная часть аппаратных средств система BIOS в ПК реализована в виде одной микросхемы, установленной на материнской плате компьютера. Большинство современных видеоадаптеров и контроллеров-накопителей имеют собственную систему BIOS, которая дополняет системную BIOS.

Как программный модуль ОС система BIOS содержит программу тестирования при включении питания компьютера POST (Power On Self Test – самотестирование при включении питания компьютера). При запуске этой программы тестируются основные компоненты компьютера. Если при подаче питания компьютера возникают проблемы, т. е. BIOS не может выполнить начальный тест, то извещение об ошибке будет выглядеть как последовательность звуковых сигналов.

В «неизменяемой» памяти CMOS RAM хранится информация о конфигурации компьютера (количестве памяти, типах накопителей и др.). Именно в этой информации нуждаются программные модули системы BIOS. Данная память выполнена на основе определенного типа CMOS-структур (CMOS – Complementary Metal Oxide Semiconductor), которые характеризуются малым энергопотреблением. Память CMOS энергонезависима, так как питается от аккумулятора, расположенного на системной плате, или батареи гальванических элементов, смонтированной на корпусе системного блока.

**Аббревиатура BIOS означает Basic Input/Output System** - базовая система ввода-вывода. Системный BIOS является программой самого нижнего уровня - он действует как интерфейс между аппаратными средствами (особенно чипсетом и процессором) и операционной системой. BIOS обеспечивает доступ к аппаратным средствам PC и позволяет разработать операционные системы более высокого уровня (DOS, Windows 95 и др), с помощью которых пользователь запускает приложения. BIOS также отвечает за управление параметрами аппаратных средств, за загрузку PC при включении питания или нажатии кнопки сброса, а также за другие системные функции.

Новые компьютеры используют прошивку UEFI вместо традиционного BIOS. Обе эти программы – примеры ПО низкого уровня, запускающегося при старте компьютера перед тем, как загрузится операционная система. UEFI – более новое решение, он поддерживает жёсткие диски большего объёма, быстрее грузится, более безопасен – и, что очень удобно, обладает графическим интерфейсом и поддерживает мышь.



### 38. Синхронный и асинхронный ввод/вывод.

Существует два типа синхронизации ввода-вывода: **синхронный ввод-вывод** и **асинхронный ввод-вывод**. Асинхронный ввод-вывод также называется перекрывающимся вводом-выводом. В **синхронном файловом вводе-выводе** поток запускает операцию ввода-вывода и сразу же переходит в состояние ожидания, пока запрос ввода-вывода не завершится. Поток, выполняющий **асинхронный файловый ввод-вывод**, отправляет запрос ввода-вывода в ядро путем вызова соответствующей функции. Если запрос принимается ядром, вызывающий поток продолжает обработку другого задания, пока ядро не сигнализирует потоку о завершении операции ввода-вывода. Затем он прерывает текущее задание и при необходимости обрабатывает данные операции ввода-вывода. Процесс синхронизации двух типов:

В ситуациях, когда запрос ввода-вывода занимает много времени, например, обновление или резервное копирование большой базы данных или медленное соединение связи, асинхронный ввод-вывод обычно является хорошим способом оптимизации эффективности обработки. Однако для относительно быстрых операций ввода-вывода затраты на обработку запросов ввода-вывода ядра и сигналов ядра могут сделать асинхронные операции ввода вывода менее полезными, особенно если необходимо выполнить множество быстрых операций ввода-вывода. В этом случае лучше использовать синхронные операции ввода-вывода. Механизмы и сведения о реализации выполнения этих задач зависят от типа используемого дескриптора устройства и конкретных потребностей приложения. Другими словами, обычно существует несколько способов решения проблемы.

### 39. Способы обмена с внешними устройствами, их программная реализация. Драйверы.

**Существует два основных способа обмена с внешними устройствами:**

- 1) Прямой доступ к памяти.
- 2) Программно-управляемый ввод-вывод.

**Прямой доступ к памяти:**

Прямой доступ к памяти (DMA - Direct Memory Access) является способом обмена данными между внешним устройством и памятью без участия процессора и предназначен в основном для устройств, обменивающихся большими блоками данных с оперативной памятью. Инициатором обмена всегда выступает внешнее устройство. Процессор инициализирует контроллер DMA, и далее обмен выполняется под управлением контроллера. Если выбранный режим обмена не занимает всей пропускной способности шины, во время операций DMA процессор может продолжать работу.

**Программно-управляемый ввод-вывод:**

Программно-управляемый ввод-вывод означает обмен данными с внешними устройствами с использованием команд процессора. Передача данных происходит через регистры процессора и при этом в конечном счете может реализовываться обмен собственно с процессором, обмен внешнего устройства с памятью, обмен между внешними устройствами.

Процессоры x86 имеют отдельную адресацию памяти и портов ввода-вывода и соответственно ввод-вывод может быть отображен либо в пространство ввода-вывода, либо в пространство оперативной памяти (memory-mapped I/O). В последнем случае адрес памяти декодируется во внешнем устройстве и для выполнения ввода-вывода могут быть использованы все команды обращения к памяти.

Каждый адресуемый элемент адресного пространства ввода-вывода именуется портом ввода, портом вывода или портом ввода-вывода. Для обращения к портам предназначены четыре основные команды процессора: **In** (ввод в порт), **Out** (вывод из порта), **Insb** (ввод из порта в элемент строки памяти) и **Outsb** (вывод элемента из строки памяти). Последние две строковые команды ввода-вывода используются для быстрой пересылки блоков данных между портом и памятью в случае последовательно расположенных адресов портов в внешнем устройстве.

**Драйвер** - компьютерное программное обеспечение, с помощью которого другое программное обеспечение (операционная система) получает доступ к аппаратному обеспечению некоторого устройства. Обычно с операционными системами поставляются драйверы для ключевых компонентов аппаратного обеспечения, без которых система не сможет работать. Однако для некоторых устройств (таких, как видеокарта или принтер) могут потребоваться специальные драйверы, обычно предоставляемые производителем устройства.

#### 40. Иерархия драйверов. Многослойная модель драйвера.

Драйверы устройств в компьютерных системах обычно организованы в иерархическую структуру, или же в многослойная модель. Это означает, что драйверы устройств работают на разных уровнях системы, в зависимости от их функций и взаимодействия с аппаратным обеспечением.

Многослойная модель драйвера представляет собой подход, в котором драйверы разбиты на отдельные слои, каждый из которых выполняет определенные функции. Например, верхние слои модели могут предоставлять абстракцию устройства и управлять его конфигурацией, а нижние слои могут обрабатывать непосредственное взаимодействие с аппаратурой. Многослойная модель позволяет разделять ответственность между различными частями драйвера, облегчает его разработку, тестирование и поддержку.

**В общем виде иерархия драйверов может быть представлена следующим образом:**

1) **Уровень ядра (Kernel Level):** Это самый низкий уровень, где драйверы взаимодействуют непосредственно с аппаратным обеспечением. Драйверы на этом уровне обычно управляют такими вещами, как прерывания процессора, доступ к памяти и другие низкоуровневые функции.

2) **Уровень аппаратного обеспечения (Hardware Level):** На этом уровне драйверы управляют конкретными устройствами, такими как диски, сетевые карты и т.д. Они обеспечивают интерфейс между аппаратным обеспечением и операционной системой.

3) **Уровень операционной системы (OS Level):** На этом уровне драйверы обеспечивают функциональность, которую операционная система может использовать для взаимодействия с аппаратным обеспечением. Это включает в себя такие вещи, как управление файловой системой, сетевые операции и т.д.

4) **Уровень приложений (Application Level):** На этом уровне драйверы обеспечивают интерфейс, который приложения могут использовать для взаимодействия с аппаратным обеспечением через операционную систему. Это включает в себя такие вещи, как API для графических операций, звуковые драйверы и т.д.

Такая многоуровневая организация позволяет маскировать аппаратные детали, упрощает разработку и делает драйвер более переносимым. Использование стандартных протоколов и API уровней также повышает совместимость с различными ОС.

#### 41. Унифицированная модель драйвера

**Унифицированная модель драйвера (Unified Driver Model, UDM)** - это концепция и подход, направленные на создание единой модели программного драйвера, которая может быть использована на различных платформах и с разными устройствами. Цель UDM состоит в том, чтобы упростить разработку драйверов, повысить их совместимость и обеспечить более эффективное использование ресурсов.

**В унифицированной модели драйвера ключевыми элементами являются:**

1. **Единый интерфейс:** UDM определяет единый интерфейс или набор API, который драйвер должен использовать для взаимодействия с устройством и операционной системой. Этот интерфейс предоставляет абстракцию от конкретных деталей устройства и позволяет разработчикам создавать драйверы, которые могут работать на разных платформах без изменений.

2. **Переносимость:** Унифицированная модель драйвера стремится к созданию драйверов, которые могут быть перенесены на различные операционные системы. Это позволяет разработчикам создавать единую версию драйвера, которая может быть использована на Windows, Linux, macOS и других платформах.

3. **Абстракция устройства:** UDM предоставляет абстракцию устройства, которая скрывает конкретные детали и особенности каждого устройства. Драйвер взаимодействует с абстрактным представлением устройства через единый интерфейс, что упрощает разработку и обслуживание драйвера.

4. **Общая функциональность:** Унифицированная модель драйвера обычно предоставляет общую функциональность, необходимую для работы с устройством, такую как управление памятью, вводом-выводом, прерываниями и другими аппаратными возможностями. Это позволяет разработчикам использовать общий код для реализации базовых функций, что повышает эффективность разработки и снижает возможность ошибок.

5. **Обратная совместимость:** Унифицированная модель драйвера также обычно поддерживает обратную совместимость с существующими драйверами. Это означает, что новые версии операционной системы будут поддерживать старые драйверы, разработанные с использованием предыдущих версий UDM, что обеспечивает плавный переход и минимизирует проблемы совместимости.

В итоге, унифицированная модель драйвера представляет собой концепцию и инфраструктуру, которая способствует разработке драйверов, обеспечивает их совместимость и повышает эффективность использования ресурсов. Это имеет важное значение для создания стабильных и совместимых систем, где драйверы играют важную роль в обеспечении работы устройств и операционной системы.

## 42. Обработка прерываний, исключений, ловушек. Типы прерываний.

**Обработчик прерываний** — специальная процедура, вызываемая по прерыванию для выполнения его обработки. Обработчики прерываний могут выполнять множество функций, которые зависят от причины, которая вызвала прерывание.

Обработчики вызываются либо по **аппаратному прерыванию** (внешние устройства, по типу мышь, клавиатура), либо по **программному** (инструкция в самой программе), и соответственно обычно предназначены для взаимодействия с устройствами или для осуществления вызова функций операционной системы.

На современных ПК обработчики основных аппаратных и программных прерываний находятся в памяти BIOS. Современная операционная система, во время своей загрузки, заменяет эти обработчики своими. При загрузке драйверов устройств операционная система распределяет управление обработкой прерывания между ними. В операционных системах семейства Windows программные прерывания используются для вызовов многих API функций. В ассемблере x86 прерывание вызывается командой INT (Например, INT 16h – прерывание для взаимодействия с клавиатурой).

### **Главные механизмы прерывания:**

#### **1) Распознавание прерывания**

На данном этапе устанавливается факт прерывания, запоминается состояние прерванного процесса.

#### **2) Передача управления обработчику прерывания**

На данном этапе происходит передача управления обработчику и сама обработка прерывания.

#### **3) Корректное возвращение к прерванной программе**

На данном этапе происходит восстановление информации, относящейся к прерванной программе, а также возврат в прерванную программу.

### **Обработчики делятся следующие типы:**

#### **1) Высокоприоритетные обработчики прерываний (ВОП)**

К ним предъявляются жёсткие требования: малое время на выполнение, малое количество операций, разрешенных к выполнению, особая надежность, так как ошибки, допущенные во время выполнения, могут обрушить операционную систему, которая не может корректно их обработать. Поэтому ВОП обычно выполняют минимально необходимую работу.

#### **2) Низкоприоритетные обработчики прерываний (НОП)**

НОП завершает обработку прерывания. НОП либо имеет собственный поток для обработки, либо заимствует на время обработки поток из системного пула. Эти потоки планируются наравне с другими, что позволяет добиться более гладкого выполнения процессов. НОП выполняется с гораздо менее жесткими ограничениями по времени и ресурсам, что облегчает программирование и использование драйверов.

### **Типы прерываний:**

1) **Асинхронные (или внешние)** – события, которые исходят от внешних аппаратных устройств (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши.

2) **Синхронные (или внутренние)** – события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памяти.

3) **Программные (частный случай внутреннего прерывания)** – инициируются исполнением специальной инструкции в коде программы. Программные прерывания, как правило, используются для обращения к функциям встроенного программного обеспечения.

Термин ловушка применяется для внутренних прерываний.

**Обработка исключений** – механизм в языках программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (*исключения*), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Примеры исключительных ситуаций: деление на ноль, ошибка считывания данных с устройства, отказ в выделении памяти, аварийное отключение питания.

**Выделяют 2 вида исключительных ситуаций:**

**1) Синхронные**

Могут возникнуть только в заранее известных точках программы. Например, известно где потребуются выделить динамическую память, и система может её не дать, такая ситуация подлежит обработке.

**2) Асинхронные**

Могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Например, аварийный отказ питания, но при наличии резервного питания данную ситуацию можно обработать.

### **43. Обработчики аппаратных прерываний, правила написания программного кода обработчиков**

Обработчики аппаратных прерываний - это функции, которые вызываются операционной системой или ядром, когда происходит заданное аппаратное прерывание на компьютере. Эти функции обрабатывают прерывание и выполняют необходимые действия для восстановления функционирования системы.

Правила написания программного кода обработчиков аппаратных прерываний зависят от конкретной архитектуры компьютера и операционной системы, но есть общие рекомендации:

1. Обработчик должен быстро выполняться, так как он работает в контексте ядра и занимает ресурсы процесса, который был прерван.
2. Напишите обработчик таким образом, чтобы он выполнял только необходимые операции, которые связаны с обработкой прерывания.
3. Используйте только безопасный код в обработчике, чтобы избежать возможных ошибок, которые могут привести к краху системы.
4. Не используйте блокировку в обработчике, так как это может привести к длительной задержке системы и даже к краху.
5. Используйте эффективные алгоритмы обработки прерываний, чтобы минимизировать временные задержки и затраты ресурсов.

В целом, при написании обработчиков аппаратных прерываний необходимо быть внимательным к нюансам конкретной системы и руководствоваться общими правилами безопасности и эффективности.



#### 44. API- поддержка обработки прерываний и событий для проектирования приложений

Программное обеспечение может столкнуться с различными событиями, такими как нажатия клавиш, сетевые запросы, изменения состояния устройств и другие. Вместо активного ожидания наступления этих событий, приложения могут зарегистрировать обработчики событий и прерываний, чтобы быть проактивными в их обработке. Это позволяет эффективно использовать ресурсы системы и реагировать на события в режиме реального времени.

API поддержки обработки прерываний и событий предоставляет набор функций и интерфейсов, которые позволяют разработчикам регистрировать обработчики событий, настраивать их параметры и обрабатывать полученные данные или действия.

Преимущества использования API поддержки обработки прерываний и событий заключаются в следующем:

- **Асинхронная обработка:** Приложения могут эффективно обрабатывать события, не блокируя основной поток выполнения. Это позволяет предотвратить задержки и повысить отзывчивость приложения.
- **Гибкость:** Разработчики имеют возможность выбирать, какие события они хотят обрабатывать, и настраивать обработчики событий согласно своим потребностям.
- **Расширяемость:** Поддержка обработки прерываний и событий позволяет легко добавлять новые функции и интегрировать внешние системы или устройства, которые могут генерировать события.
- **Масштабируемость:** Архитектура, основанная на обработке событий, облегчает создание масштабируемых систем, так как приложения могут обрабатывать события асинхронно и распределять нагрузку между различными обработчиками.

Одним из примеров таких API является Windows API: Операционная система Windows предоставляет API для обработки прерываний и событий. Например, функции RegisterDeviceNotification и WndProc позволяют регистрировать обработчики событий устройств, таких как вставка или удаление USB-устройств. Это позволяет приложению реагировать на изменения в подключенных устройствах.



## **45. Современные средства разработки ПО (SDK, IDE, Toolkits). Инструментальные средства ОС для трассировки, анализа и отладки кода.**

Современные средства разработки программного обеспечения (ПО) включают в себя широкий спектр инструментов и технологий, которые значительно упрощают и ускоряют процесс разработки. Среди них можно выделить следующие:

- **Интегрированные среды разработки (IDE)** - это программные пакеты, которые объединяют в себе несколько инструментов, необходимых для создания и отладки приложений. Как правило, IDE содержат текстовый редактор, компилятор, отладчик, а также другие инструменты, такие как графический дизайнер интерфейсов, базы данных и т.д. Примеры популярных IDE: Visual Studio, Eclipse, IntelliJ IDEA, NetBeans.

- **Комплекты разработчика ПО (SDK)** - это наборы ПО и инструментов, необходимых для разработки приложений на определенной платформе или для определенного языка программирования. SDK может включать в себя компиляторы, библиотеки, инструменты отладки, документацию и т.д. Например, Android SDK, iOS SDK, Java SDK.

- **Инструментарий разработки пользовательского интерфейса (UI Toolkits)** - это библиотеки и инструменты, используемые для создания графических интерфейсов пользователей. Они снижают сложность и время разработки интерфейса, позволяют создавать качественные визуальные элементы и обеспечивают совместимость с различными платформами. Примеры UI Toolkits: Qt, wxWidgets, JavaFX.

В ОС также есть встроенные инструментальные средства для трассировки, анализа и отладки кода. Некоторые из них:

- **Профилировщик (Profiler)** - это инструмент для измерения производительности и нахождения узких мест в программе. Позволяет определить точки, где программа затрачивает наибольшее количество времени, и улучшить ее производительность. Примеры: Visual Studio Profiler, Xcode Instruments.

- **Отладчик (Debugger)** - это инструмент для устранения ошибок в коде, позволяет запускать программу в режиме отладки шаг за шагом, задавать точки останова и искать ошибки. Примеры: gdb (Linux), WinDbg (Windows), lldb (macOS/iOS).

- **Монитор памяти (Memory Profiler)** - это инструмент для обнаружения утечек памяти, позволяет отслеживать использование оперативной памяти в программе и находить утечки. Примеры: Visual Studio Memory Profiler, Valgrind.

- **Трассировщик (Tracer)** - это инструмент для изучения работы программы и ее взаимодействия с внешней средой. Позволяет записывать и анализировать последовательность вызовов функций в программе. Примеры: strace (Linux), Process Monitor (Windows), DTrace (macOS)