**1)** **Вычислительная система. Обобщенная структура и иерархическое представление. Понятие ресурса и его виртуализация.**

Вычислительная система рассматривается как многослойная структура из нескольких функциональных уровней. Каждый уровень определяется данными, выполняемыми функциями и результатами.

Самый нижний — аппаратура (микроархитектура). Каждый более вышестоящий для выполнения своих функций может запрашивать сервис более низкого уровня. Результат выполнения функций нижнего уровня передается на более высокий уровень.

**Иерархическое представление**:

Приложения пользователей

Уровень пользователя

Системные вызовы

Аппаратно независимые модули

Аппаратно зависимые модули

Уровень ядра

Аппаратура

При этом имеет место изоляция несмежных уровней друг от друга.

**Обобщенная структура вычислительной системы** (взгляд IT-специалиста)

1 Прикладные программы, GUI

2 Пакеты прикладных подпрограмм

Утилиты

{

3 Системы программирования: Отладчики, Компоновщики, Редакторы, Трансляторы

3 Библиотеки подпрограмм

4 Системный интерфейс

5 Операционная система

}

6 Аппаратура вычислительной системы

Ресурс — всякий потребляемый объект, независимо от формы его существования, обладающий некоторой практической ценностью для потребителя.

Вычислительный ресурс — возможности, обеспечиваемые компонентами вычислительной системы, расходуемые или занимаемые в процессе её работы.

Одним из важнейших свойств ресурса является «реальность существования».

В этом смысле ресурсы разделяют на физические и виртуальные (мнимые).

Под физическим понимают ресурс, который реально существует и при распределении его между пользователями обладает всеми присущими ему физическими характеристиками (физическая память, доступ к физическому каналу).

Виртуализация — предоставление набора вычислительных ресурсов или

их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию вычислительных процессов, выполняемых на одном физическом ресурсе.

Примером использования виртуализации является возможность запуска нескольких операционных систем на одном компьютере.

**2) Операционная система. Классификация и структура.**

Операционная система — комплекс управляющих и обрабатывающих программ, который выступает как интерфейс между аппаратурой компьютера и пользователем; предназначен для эффективного и безопасного использования ресурсов ВС.

Любой из компонентов прикладного программного обеспечения обязательно работает под управлением ОС. Ни один из компонентов программного обеспечения, за исключением самой ОС, не имеет непосредственного доступа к аппаратуре компьютера. Пользователи взаимодействуют со своими программами через интерфейс ОС. Любые их команды, прежде чем попасть в прикладную программу, сначала проходят через ОС.

Классификация: по сфере применения: мобильная, настольная(пк), серверная, встраеваемоя, по числу пользователей: однопользовательская, многопользовательская,

**по архитектуре**: **Монолитные ОС (Monolithic OS)**: Вся операционная система работает в режиме ядра. Примеры: Linux, UNIX. **Микроядерные ОС (Microkernel OS)**: Минимальное ядро, основные функции вынесены в пользовательское пространство. Примеры: QNX, Minix. **Гибридные ОС (Hybrid OS)**: Сочетают элементы монолитных и микроядерных архитектур. Примеры: Windows NT, macOS.

Структура операционной системы:

**- Ядро** взаимодействует с аппаратным обеспечением, предоставляя интерфейсы для остальных компонентов ОС.

**- Системные библиотеки** используют функции ядра для выполнения системных вызовов.

**- Системные утилиты** и **оболочка** предоставляют пользователям и приложениям удобные средства для работы с системой.

**-Средства безопасности** обеспечивают защиту данных и контроль доступа на всех уровнях операционной системы.

**3) Мультизадачность и многопоточность. Концепция процесса. Диаграмма состояний и операции над процессами.**

Отдельное процессорное ядро в один момент времени может исполнять только одну программу. Однако современные вычислительные системы предъявляют более широкие требования к обработке задач. Чтобы удовлетворить эти требования, современные процессоры обладают мультизадачными возможностями.

#### Мультизадачность

Процессор выполняет одну программу (процесс или задачу) в течение определенного времени (обычно микросекунды). После этого операционная система переключает процессор на выполнение другой программы. При этом все регистры текущей программы (ее состояние и контекст) сохраняются, и через некоторое время управление снова передается этой программе. Благодаря этому процесс переключения остается незаметным для программы, она продолжает свою работу без перебоев.

#### Механизмы защиты

Чтобы программы не могли нарушить работу системы или других программ, предусмотрены механизмы защиты на основе уровней привилегий. Уровни привилегий определяют степень использования ресурсов процессора. Всего таких уровней четыре, и они имеют номера от 0 до 3:

**- Уровень 0**: Самый высокий уровень привилегий, используется ядром операционной системы.

**- Уровень 1 и 2**: Используются для системных задач и драйверов устройств.

**- Уровень 3**: Самый низкий уровень привилегий, предназначен для пользовательских приложений.

Эти механизмы обеспечивают стабильность и безопасность работы системы, предотвращая несанкционированный доступ и возможные ошибки со стороны программ.

Процесс — это находящаяся под управлением ОС совокупность некоторого набора исполняющихся инструкций, ассоциированных с процессом ресурсов ВС и текущего момента выполнения. Процесс не может сам перейти из одного состояния в другое. Изменением состояния процессов занимается ОС, совершая над ними операции:

-Рождение

-Приостановка — запуск

-Блокирование — разблокирование

- Изменение приоритета

Каждый процесс может находиться, как минимум, в двух состояниях:

- процесс исполняется;

- процесс не исполняется.

Процесс, находящийся в состоянии «процесс исполняется», может через некоторое время завершиться или быть приостановлен операционной системой и переведен в состояние «процесс не исполняется». Приостановка происходит по одной из двух причин:

1) для его дальнейшей работы потребовалось возникновение какого-либо события

2) истек временной интервал, отведенный операционной системой для работы этого процесса.

После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии «процесс не исполняется», и переводит его в состояние «процесс исполняется».

Процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов.

Чтобы это учесть, состояние «процесс не исполняется» разбивается на два новых:

- готовность;

- ожидание.

**4) Структуры управления процессами. Контексты. Переключение контекста.**

### Process Control Block и контекст процесса

#### Process Control Block (PCB)

Для выполнения операций над процессами операционная система представляет каждый процесс в виде структуры данных, называемой **Process Control Block (PCB)**. PCB содержит информацию, специфическую для каждого процесса:

* **Состояние процесса**: Текущее состояние процесса (готов, выполняется, ждет и т.д.).
* **Программный счетчик**: Адрес следующей команды, которую нужно выполнить.
* **Содержимое регистров процессора**: Значения регистров в момент переключения контекста.
* **Данные для планирования**: Приоритет процесса, размер и расположение адресного пространства и т.д.
* **Учетные данные**: Идентификационный номер процесса, пользователь, который инициировал процесс, общее время использования процессора и т.д.
* **Информация об устройствах ввода-вывода**: Список устройств, закрепленных за процессом, и таблица открытых файлов.

Во многих операционных системах информация о процессе хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут содержать дополнительную информацию или лишь часть описанной выше.

PCB содержит всю необходимую информацию для выполнения операций над процессом, представляя его модель для операционной системы. Любая операция, производимая ОС над процессом, вызывает изменения в PCB.

#### Контекст процесса

Контекст процесса можно разделить на две части:

* **Регистровый контекст**: Содержимое всех регистров процессора, включая значение программного счетчика.
* **Системный контекст**: Вся остальная информация, необходимая для управления процессом.

Знание регистрового и системного контекстов достаточно для управления поведением процесса в операционной системе.

#### Пользовательский контекст

Пользовательский контекст включает код и данные, находящиеся в адресном пространстве процесса. Операционной системе не важно, чем именно занимается процесс, а вот для пользователя важны содержимое адресного пространства и результаты выполнения процесса.

Контекст процесса — это совокупность регистрового, системного и пользовательского контекстов. В любой момент времени процесс полностью характеризуется своим контекстом.

#### Задача в архитектуре x86 (IA32)

В контексте архитектуры x86 (IA32) используется термин **задача (task)**, который является эквивалентом понятия процесс. Задача — это последовательность команд, выполняющаяся в своем окружении. Основные параметры, характеризующие окружение задачи:

* **Состояние регистров общего назначения**: EAX, EBX, ECX, EDX, ESI, EDI.
* **Состояние селекторных (сегментных) регистров**: DS, CS, SS, ES, FS, GS.
* **Адресное пространство**: Характеризуется регистром CR3.
* **Состояние регистров математического сопроцессора и расширений**: MMX, SSE, XMM, AVX и т.д.
* **Карта портов ввода-вывода**.

#### Многозадачность

Поскольку количество задач (процессов) обычно превышает количество процессоров (ядер), многозадачность реализуется через быстрое переключение задач. Это создает ощущение одновременного выполнения всех задач.

**5) Планирование процессов. Долгосрочное и краткосрочное планирование. Критерии и требования к алгоритмам.**

Планирование процессов в операционной системе - это механизм управления очередью процессов, ожидающих выполнения. Этот механизм обеспечивает эффективное распределение ресурсов процессора между процессами. Существует несколько уровней планирования:

1. **Долгосрочное планирование (Long-Term Scheduling)**
2. **Краткосрочное планирование (Short-Term Scheduling)**
3. **Среднесрочное планирование (Medium-Term Scheduling)** (иногда рассматривается отдельно, но не так часто упоминается, как первые два)

#### Долгосрочное планирование (Long-Term Scheduling)

Долгосрочное планирование отвечает за выбор процессов из пула задач, находящихся на диске, и их загрузку в оперативную память для выполнения. Это планирование определяет, какие процессы и когда будут допущены в систему для выполнения.

* **Цель**: Контролировать степень мультипрограммирования, то есть количество процессов, находящихся в оперативной памяти.
* **Частота выполнения**: Выполняется нечасто, в зависимости от поступления новых задач и завершения существующих процессов.
* **Критерии**:
  + **Балансировка нагрузки**: Систематическое распределение различных типов задач (например, CPU-bound и I/O-bound) для обеспечения эффективного использования ресурсов.
  + **Стабильность системы**: Поддержание определенного количества процессов в системе для предотвращения перегрузки.

#### Краткосрочное планирование (Short-Term Scheduling)

Краткосрочное планирование занимается выбором одного из готовых к выполнению процессов и его назначением на выполнение на процессоре. Этот тип планирования также известен как диспетчеризация (dispatching).

* **Цель**: Обеспечить максимально эффективное использование процессора и быстроту отклика системы.
* **Частота выполнения**: Выполняется очень часто, на уровне миллисекунд, чтобы оперативно переключать процессорное время между процессами.
* **Критерии**:
  + **Время отклика**: Время, необходимое системе для отклика на запрос пользователя.
  + **Время выполнения (turnaround time)**: Время, которое проходит с момента поступления задачи в систему до её завершения.
  + **Производительность (throughput)**: Количество процессов, завершаемых за единицу времени.
  + **Использование процессора**: Процент времени, в течение которого процессор занят выполнением задач.

#### Критерии и требования к алгоритмам планирования

Алгоритмы планирования процессов должны удовлетворять определенным критериям и требованиям для обеспечения эффективного и справедливого распределения ресурсов.

1. **Справедливость (Fairness)**: Каждый процесс должен получать справедливую долю процессорного времени, избегая ситуаций, когда одни процессы получают слишком много времени в ущерб другим.
2. **Эффективность использования процессора (CPU Utilization)**: Максимизация использования процессора, чтобы процессор не простаивал без работы.
3. **Время отклика (Response Time)**: Минимизация времени отклика для интерактивных пользователей, чтобы система казалась отзывчивой.
4. **Среднее время выполнения (Average Turnaround Time)**: Минимизация среднего времени выполнения процессов.
5. **Среднее время ожидания (Average Waiting Time)**: Минимизация среднего времени, которое процессы проводят в очереди ожидания процессорного времени.
6. **Производительность (Throughput)**: Максимизация количества выполненных процессов за единицу времени.

**6) Алгоритмы планирования. First-Come, First-Served (FCFS). Round Robin (RR). Shortest-Job-First (SJF).**Смотри пункт 5 допом

### Основные алгоритмы планирования

1. **FCFS (First-Come, First-Served)**:

**Описание**: Процессы обслуживаются в порядке поступления.

**Преимущества**: Простота реализации.

**Недостатки**: Может привести к проблемам с производительностью из-за феномена "конвоя" (Convoy Effect).

1. **SJF (Shortest Job First)**:

**Описание**: Процессы с наименьшим ожидаемым временем выполнения обслуживаются первыми.

**Преимущества**: Оптимальное среднее время выполнения.

**Недостатки**: Трудности в оценке времени выполнения процессов и проблема "голодания" (starvation) для долгих процессов.

1. **Round Robin (RR)**:

**Описание**: Каждому процессу выделяется фиксированный квант времени. Процессы обслуживаются по кругу.

**Преимущества**: Хорошо подходит для систем с интерактивными пользователями.

**Недостатки**: Производительность сильно зависит от выбора величины кванта времени.

**7) Алгоритмы планирования. Гарантированное планирование. Приоритетное планирование.**

Гарантированное планирование (Guaranteed Scheduling) - это стратегия планирования, при которой каждому процессу гарантирована определенная доля процессорного времени. Это обеспечивает предсказуемость выполнения задач и гарантирует, что ни один процесс не будет "голодать" или занимать все ресурсы процессора.

#### Особенности:

1. **Гарантированные квоты времени**: Каждый процесс получает определенное количество времени процессора для выполнения своей работы.
2. **Предсказуемость выполнения**: Поскольку каждому процессу предоставляется гарантированная квота времени, можно точно определить, когда процесс будет завершен.
3. **Предотвращение голодания**: Ни один процесс не остается без процессорного времени из-за наличия гарантированных квот.

#### Примеры алгоритмов гарантированного планирования:

1. **Rate-Monotonic Scheduling (RMS)**:

В этом методе процессы планируются на основе их периодов выполнения (частоты).

Процессы с более короткими периодами выполнения получают более высокий приоритет и гарантированно больше времени CPU.

1. **Earliest Deadline First (EDF)**:

Процессы планируются на основе их сроков завершения.

Процесс с самым ранним сроком завершения получает высший приоритет и выполняется первым.

### Приоритетное планирование

Приоритетное планирование (Priority Scheduling) - это стратегия планирования, при которой процессы обслуживаются в порядке их приоритетов. Процесс с более высоким приоритетом получает процессорное время в ущерб процессу с более низким приоритетом.

#### Особенности:

1. **Приоритеты процессов**: Каждому процессу назначается определенный приоритет, определяющий его важность или срочность.
2. **Процесс с наивысшим приоритетом получает процессорное время**: Процесс с более высоким приоритетом будет выполнен раньше, чем процессы с более низкими приоритетами.
3. **Возможность голодания**: Если процессы с низким приоритетом постоянно поступают, они могут столкнуться с проблемой "голодания", когда им не уделяется достаточного процессорного времени.

#### Примеры алгоритмов приоритетного планирования:

1. **Статическое приоритетное планирование**:

Каждому процессу назначается статический приоритет в момент его создания или в момент планирования.

Примеры: Fixed Priority Preemptive Scheduling (FPPS), Fixed Priority Non-Preemptive Scheduling (FPNS).

1. **Динамическое приоритетное планирование**:

Приоритеты процессов изменяются в процессе выполнения в зависимости от их поведения и условий системы.

Примеры: Aging, Priority Inheritance, Priority Ceiling Protocol.

### **8) Алгоритмы планирования. Многоуровневые очереди (Multilevel Queue). Многоуровневые очереди с обратной связью (Multilevel Feedback Queue).**

### Многоуровневые очереди (Multilevel Queue)

Многоуровневая очередь - это стратегия планирования, при которой процессы разделены на несколько групп (очередей) в зависимости от их характеристик или типа. Каждая очередь может иметь свой собственный алгоритм планирования и приоритет.

#### Особенности:

1. **Разделение процессов по уровням приоритета**: Процессы классифицируются на несколько групп в зависимости от их приоритета или характеристик (например, CPU-bound и I/O-bound).
2. **Разные алгоритмы планирования для разных очередей**: Разные группы процессов могут обслуживаться разными алгоритмами планирования в зависимости от их характеристик и требований.
3. **Адаптивность к различным типам процессов**: Многоуровневая очередь позволяет эффективно управлять различными типами процессов, оптимизируя их выполнение.

#### Примеры алгоритмов планирования для многоуровневых очередей:

1. **Приоритетное планирование для каждой очереди**: Каждая очередь имеет свой уровень приоритета, и процессы в каждой очереди планируются на основе их приоритетов.
2. **Round Robin для высокоприоритетных очередей**: Процессы в высокоприоритетных очередях могут использовать алгоритм Round Robin для обеспечения справедливого распределения процессорного времени.
3. **First-Come, First-Served для низкоприоритетных очередей**: Процессы в низкоприоритетных очередях могут использовать алгоритм FCFS для простоты реализации.

### Многоуровневые очереди с обратной связью (Multilevel Feedback Queue)

Многоуровневые очереди с обратной связью - это расширение многоуровневой очереди, которое позволяет процессам изменять свои приоритеты и переходить между разными уровнями очередей в зависимости от их поведения и времени выполнения.

#### Особенности:

1. **Динамическое изменение приоритетов**: Приоритеты процессов могут изменяться в процессе выполнения в зависимости от их поведения и времени выполнения.
2. **Переход между уровнями очередей**: Процессы могут перемещаться между разными уровнями очередей в зависимости от их текущего состояния и требований системы.
3. **Адаптивность к различным условиям и нагрузке**: Многоуровневые очереди с обратной связью позволяют системе эффективно реагировать на изменения в нагрузке и требованиях.

#### Примеры алгоритмов планирования для многоуровневых очередей с обратной связью:

1. **Алгоритм с возрастающими приоритетами (Aging)**: Приоритеты процессов увеличиваются с течением времени, чтобы предотвратить "голодание" процессов с низким приоритетом.
2. **Алгоритм с квантованием времени (Quantum Time)**: Процессы в каждой очереди имеют фиксированный квант времени, после которого они перемещаются на следующий уровень очереди.
3. **Алгоритм с откатом приоритета (Priority Rollback)**: Приоритет процесса уменьшается, если он длительное время ожидает выполнения, и увеличивается, когда он активно используется.

**9) Логическая и физическая память. Основные функции ОС по управлению памятью. Простые схемы управления памятью.**

#### Логическая память

**Виртуальные адреса**: Программы используют виртуальные адреса, которые ОС переводит в физические.

**Изоляция процессов**: Логическая память изолирует процессы друг от друга.

**Большое адресное пространство**: Виртуальная память позволяет программам работать с большим адресным пространством, чем доступная физическая память.

#### Физическая память

**Физические адреса**: Это реальные адреса в оперативной памяти.

**Прямая адресация**: Процессор и контроллеры памяти напрямую работают с физическими адресами.

### Основные функции ОС по управлению памятью

1. **Размещение памяти**:

**Статическое**: Память выделяется при загрузке программы.

**Динамическое**: Память выделяется и освобождается по мере необходимости.

1. **Защита памяти**:

**Изоляция процессов**: Каждый процесс имеет своё адресное пространство.

**Аппаратная защита**: Использование таблиц страниц и регистров защиты.

1. **Отображение памяти**:

**Таблицы страниц**: Преобразование виртуальных адресов в физические.

**Сегментирование**: Разделение памяти на сегменты.

1. **Перемещение и своппинг**:

**Перемещение (Paging)**: Разделение памяти на страницы.

**Своппинг (Swapping)**: Перемещение неактивных страниц на диск.

### Простые схемы управления памятью

1. **Однопрограммное размещение**: Вся память выделяется одному процессу.
2. **Фиксированные разделы (Fixed Partitioning)**: Память делится на фиксированные разделы, каждый раздел — один процесс.
3. **Динамическое разделение (Dynamic Partitioning)**: Память делится на разделы по потребностям процессов.
4. **Страничная организация (Paging)**: Память делится на страницы, избегая внешней фрагментации.
5. **Сегментная организация (Segmentation)**: Память делится на логические сегменты.
6. **Комбинированные схемы (Segmentation with Paging)**: Сочетание сегментации и страничной организации для гибкого управления памятью.

**10) Кооперация (взаимодействие) процессов. Категории средств межпроцессного обмена информацией.**

Процессы могут взаимодействовать друг с другом, обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории:

#### 1. Сигнальные (Signal)

* **Описание**: Передается минимальное количество информации — один бит («да» или «нет»).
* **Использование**: Предназначены для извещения процесса о наступлении какого-либо события.
* **Воздействие**: Минимальное, зависит от того, знает ли процесс, что означает полученный сигнал, и как на него реагировать. Неправильная реакция или игнорирование сигнала могут привести к проблемам.

#### 2. Канальные (Pipe)

* **Описание**: Процессы общаются через коммуникационные линии, предоставляемые операционной системой.
* **Объем информации**: Ограничен пропускной способностью линий связи.
* **Воздействие**: С увеличением объема передаваемой информации растет возможность влияния на поведение другого процесса.

#### 3. Совместно используемая память (Shared Memory)

* **Описание**: Два или более процессов могут совместно использовать определенную область адресного пространства.
* **Создание**: Совместно используемая память создается операционной системой по запросу процесса.
* **Возможность обмена**: Максимальная, как и возможность влияния на поведение другого процесса, что требует осторожности.
* **Использование**: Обмен информацией осуществляется с помощью обычных языков программирования, без необходимости в специальных системных вызовах.
* **Преимущество**: Является наиболее быстрым способом взаимодействия процессов в одной вычислительной системе.

Эти средства обмена информацией обеспечивают гибкость и позволяют выбирать подходящий способ взаимодействия процессов в зависимости от конкретных требований и условий задачи.

**11) Кооперация (взаимодействие) процессов. Логическая организация механизма передачи информации.**

#### 1. Установление связи

**Сигнальные (Signal)**: Для передачи сигнала от одного процесса к другому инициализация не требуется.

**Канальные (Pipe)**: Передача информации по каналам может потребовать первоначального резервирования линии связи для процессов, желающих обменяться данными.

**Совместно используемая память (Shared Memory)**: Для использования совместно используемой памяти различными процессами требуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства.

#### 2. Адресация

Для передачи или получения информации необходимо указать, куда ее следует передать или откуда ее можно получить. Различают два способа адресации:

**Прямая адресация**

**Симметричная прямая адресация**: Оба процесса явно указывают имена или ID своих партнеров. Это изолированная схема, при которой никакой другой процесс не может вмешаться в обмен данными.

**Асимметричная прямая адресация**: Только один из взаимодействующих процессов указывает имя своего партнера, а другой процесс может рассматривать любого процесса в качестве возможного партнера.

**Непрямая адресация**

Данные помещаются передающим процессом в промежуточный объект для хранения. Принимающий процесс извлекает данные из этого объекта, не зная, кто их передал.

Прямая адресация не требует дополнительных инициализирующих действий, в то время как при непрямой адресации инициализация может требоваться или не требоваться в зависимости от ситуации.

#### 3. Информационная валентность процессов и средств связи

**Прямая адресация**: Только одно средство связи может быть задействовано для обмена данными между двумя процессами.

**Непрямая адресация**: Один объект может использоваться более чем двумя процессами, и несколько объектов могут быть задействованы между двумя процессами.

#### 4. Направленность связи

**Однонаправленная связь**: Каждый процесс может использовать средство связи либо только для приема информации, либо только для ее передачи.

**Двунаправленная связь**: Каждый процесс может использовать связь как для приема, так и для передачи данных.

**Симплексная связь**: Однонаправленная.

**Полудуплексная связь**: Двунаправленная с поочередной передачей.

**Дуплексная связь**: Двунаправленная с возможностью одновременной передачи информации в обоих направлениях.

**12) Кооперация (взаимодействие) процессов. Особенности межпроцессной передачи информации. Буферизация, потоки и сообщения.**

Передача информации через линии связи считается безопасной и информативной, особенно по сравнению с использованием совместно используемой памяти и сигнальными средствами коммуникации.

#### Буферизация

Буферизация в линиях связи позволяет временно хранить передаваемую информацию:

**Нулевая емкость буфера**: Информация передается непосредственно без буферизации, что требует синхронизации между процессами.

**Ограниченная емкость буфера**: Буфер имеет фиксированный размер, что может привести к задержкам, если буфер полон.

**Неограниченная емкость буфера**: Теоретически возможно, но практически не всегда реализуемо из-за ограничений ресурсов.

#### Потоки данных и сообщения

**Потоки данных**: Используются для непрерывной передачи информации без структурирования.

**Сообщения**: Позволяют передавать структурированные данные с определенными заголовками и контрольной информацией.

#### Надежность и обеспечение порядка

Для обеспечения надежности и сохранения порядка данных могут применяться контрольные суммы и коды исправления ошибок (ECC).

#### Завершение связи

Важно правильно завершать связь между процессами, особенно если она была инициализирована с определенными ресурсами или состоянием.

**13) Нити исполнения. Многопоточность.**

Нити исполнения (или потоки, от англ. "threads") и многопоточность - это концепции, которые позволяют программам выполнять несколько задач параллельно. Это особенно важно для повышения эффективности и производительности современных программ и систем.

#### Нити исполнения

Нить исполнения (поток) - это наименьшая единица выполнения, которая может быть запланирована операционной системой. Потоки представляют собой подмножества процессов, которые могут выполняться параллельно, разделяя при этом ресурсы одного процесса.

##### Особенности:

1. **Разделение ресурсов**: Потоки одного процесса разделяют его ресурсы, включая память, файлы и т.д.
2. **Собственные ресурсы**: У каждого потока есть свои собственные регистры, программный счетчик и стек.
3. **Легковесность**: Потоки легче создавать и управлять ими по сравнению с процессами, так как создание потока не требует выделения отдельного адресного пространства.

#### Многопоточность

Многопоточность (multithreading) - это способность процессора или операционной системы выполнять несколько потоков одновременно.

**14) Синхронизация активностей. Достаточные условия Бернстайна**

Условия Бернстайна определяют критерии, которым должны удовлетворять два параллельно выполняемых процесса или программы, чтобы они могли выполняться параллельно без конфликтов данных. Эти условия особенно полезны при анализе программ на наличие гонок данных.

#### Условия Бернстайна:

Пусть P и Q — два процесса. Условия Бернстайна могут быть сформулированы следующим образом:

1. **Множество чтения R**:

R(P) — множество всех переменных, читаемых процессом P.

R(Q) — множество всех переменных, читаемых процессом Q.

1. **Множество записи W**:

W(P) — множество всех переменных, записываемых процессом P.

W(Q) — множество всех переменных, записываемых процессом Q.

Для того чтобы процессы P и Q могли выполняться параллельно без конфликтов данных, должны выполняться следующие условия:

1. **W(P)∩W(Q)=∅**:

Процессы не должны записывать в одни и те же переменные.

1. **W(P)∩R(Q)=∅**:

Процесс P не должен записывать в переменные, которые читает процесс Q.

1. **R(P)∩W(Q)=∅**:

Процесс Q не должен записывать в переменные, которые читает процесс P.

**15) Взаимное исключение и критическая секция.**

### Взаимное исключение (Mutual Exclusion)

Если очередность доступа к совместно используемым данным не имеет значения, проблему упорядоченного доступа к этим данным (устранение гонки данных) можно решить, обеспечив каждому процессу эксклюзивное право доступа к данным. Каждый процесс, обращающийся к общим ресурсам, исключает возможность доступа других процессов к этим ресурсам одновременно с ним, если это может привести к неопределённому поведению.

Такой метод называется взаимным исключением (mutual exclusion).

Если для получения правильных результатов важна очередность доступа к совместно используемым ресурсам, одного взаимного исключения будет недостаточно.

### Критическая секция

Важным понятием при изучении способов синхронизации процессов является критическая секция (critical section) программы.

Критическая секция — это часть программы, исполнение которой может привести к возникновению состояния гонки данных по отношению к некоторому ресурсу. Чтобы исключить эффект гонок данных, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимного исключения для критических секций программ.

Реализация взаимного исключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция выполняется как атомарная операция.

**16) Пять требований к алгоритмам организации взаимодействия процессов.**

Для эффективной организации взаимодействия процессов, которые имеют критические участки и могут проходить их в произвольном порядке, необходимо выполнение следующих условий:

1. **Программное решение**: Алгоритм должен быть реализован чисто программным способом на обычной машине, не имеющей специальных команд для взаимного исключения. Хотя такие машины сегодня редкость, предполагается, что основные инструкции языка программирования (такие как load, store, test) являются атомарными операциями.
2. **Независимость от скорости и количества процессоров**: Алгоритм не должен делать никаких предположений о относительных скоростях выполнения процессов или количестве процессоров, на которых они исполняются.
3. **Взаимоисключение (Mutual Exclusion)**: Если один процесс находится в своей критической секции, никакие другие процессы не могут быть в своих критических секциях одновременно. Это условие гарантирует, что ресурсы не будут использоваться одновременно несколькими процессами, что может привести к неопределённому поведению.
4. **Прогресс (Progress)**: Процессы, находящиеся вне своих критических секций и не собирающиеся входить в них, не должны препятствовать другим процессам вхождению в их критические секции. Если нет процессов в критических секциях, и есть процессы, желающие войти в них (находящиеся в прологе), решение о том, какой процесс войдет в свою критическую секцию, должны принимать только процессы, находящиеся в прологе. Это решение не должно приниматься бесконечно долго.

**Регулярная часть кода**: Часть кода, не связанная с критическими секциями.

**Пролог**: Часть кода, в которой процесс готовится войти в критическую секцию.

**Критическая секция**: Часть кода, в которой происходит доступ к общим ресурсам.

**Эпилог**: Часть кода, в которой процесс завершает работу в критической секции и выходит из неё.

1. **Ограниченное ожидание (Bounded Waiting)**: Не должно возникать бесконечного ожидания для входа процесса в свою критическую секцию. С момента, когда процесс запросил разрешение на вход в критическую секцию, до момента, когда он это разрешение получил, другие процессы могут пройти через свои критические секции лишь ограниченное число раз.

**17) Алгоритмы организации взаимодействия процессов. Запрет прерываний, переменнаязамок, строгое чередование и флаги готовности.**

Одним из простейших способов организации доступа к критическим секциям является запрет прерываний. Этот метод можно реализовать следующим образом:

regular\_section

interrupts\_disable(); // запретить все прерывания

critical\_section

interrupts\_enable(); // разрешить все прерывания

regular\_section

Поскольку выход процесса из состояния исполнения без его завершения осуществляется по прерыванию, никто не может вмешаться в его работу внутри критической секции. Однако этот метод имеет существенные недостатки, так как позволяет процессам пользователя управлять прерываниями во всей системе. Например, из-за ошибки или злого умысла пользователь может запретить прерывания и зациклить или завершить свой процесс, что приведет к серьезным проблемам.

Тем не менее, запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например, при обновлении содержимого PCB (Process Control Block).

### Переменная-замок (Lock Variable)

Определим переменную lock, доступную всем процессам, и присвоим ей начальное значение 0. Процесс может войти в критическую секцию, только когда значение этой переменной равно 0, одновременно изменяя её значение на 1 — закрывая замок. При выходе из критической секции процесс сбрасывает значение переменной в 0 — замок открывается.

shared int lock = 0; // свободно/занято

regular\_section

while(lock);

lock = 1; // не атомарное действие

critical\_section

lock = 0;

regular\_section

К сожалению, это решение не удовлетворяет условию взаимоисключения, так как операция while(lock); lock = 1; не является атомарной.

### Строгое чередование

Попробуем решить задачу для двух процессов — P0 и P1. Используется общая переменная turn с начальным значением 0. Если turn == 0, в критическую секцию входит P0, если turn == 1, в критическую секцию входит P1.

shared int turn = 0; // чья очередь

regular\_section\_before

while(turn != i); // ждем, пока второй процесс не выйдет

critical\_section

turn = 1 - i;

regular\_section\_after

В этом случае взаимоисключение гарантируется, и процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, ... Однако этот алгоритм не удовлетворяет условию прогресса — процессы, которые находятся вне своих критических участков и не собираются входить в них, не должны препятствовать другим процессам входить в их критические участки. Например, если turn == 0 и процесс P1 готов войти в критический участок, он не может сделать этого, если процесс P0 находится в своей регулярной секции.

### Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы не знают о состоянии друг друга. Пусть два наших процесса имеют совместно используемый массив флагов готовности:

shared int ready[2] = {0, 0};

Когда процесс k готов войти в критическую секцию, он присваивает своему элементу массива ready[i] значение 1. После выхода из критической секции он сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов ко входу или находится в ней.

ready[i] = 1;

while(ready[1-i]);

critical\_section

ready[i] = 0;

Этот алгоритм обеспечивает взаимоисключение, но может привести к тупиковой ситуации (deadlock). Если процессы одновременно выполнят присваивание ready[0] = 1 и ready[1] = 1, оба процесса будут бесконечно ждать друг друга на входе в критическую секцию.

**18) Алгоритмы организации взаимодействия процессов. Алгоритмы Петерсона и регистратуры.**

Оба процесса имеют доступ к массиву флагов готовности и к переменной очередности:

shared int ready[2] = {0, 0};

shared int turn;

ready[i] = 1;

turn = 1 - i;

while (ready[1 - i] && turn == 1 - i);

critical section

ready[i] = 0;

При исполнении пролога критической секции процесс заявляет о своей готовности выполнить критический участок (1) и одновременно предлагает другому процессу приступить к его выполнению (2).

Если оба процесса подошли к прологу практически одновременно, они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда последует после другого, что позволяет продолжить работу в критическом участке процессу, которому было сделано последнее предложение.

Этот алгоритм удовлетворяет всем пяти требованиям для корректного взаимодействия процессов.

### Алгоритм регистратуры (Bakery Algorithm)

Алгоритм Петерсона дает решение для двух процессов. Для нескольких процессов используется алгоритм регистратуры (булочной).

Каждый процесс получает талончик с номером. Процесс с наименьшим номером обслуживается следующим. Если номера талончиков равны, первым обслуживается процесс с меньшим значением имени (имена сравниваются в лексикографическом порядке).

shared enum {false, true} choosing[n]; // получает талончик

shared int number[n]; // номер талончика

do {

choosing[i] = true;

number[i] = sup(number[0], number[1], ..., number[n - 1]) + 1; // автоинкремент

choosing[i] = false;

for (j = 0; j < n; j++) {

// ждем, пока не подойдет очередь

while (choosing[j]);

// ждем, пока j получает талончик

while ((number[j] != 0) && (number[j], j) < (number[i], i));

}

// критическая секция

number[i] = 0;

// остаточная секция

} while (1);

### Объяснение

1. **Начало алгоритма**: Процесс получает талончик, присваивая себе следующий номер талончика. Этот номер больше всех текущих номеров других процессов.
2. **Ожидание очереди**: Процесс ждет, пока все другие процессы не закончат получать талончики. Если процесс имеет талончик с меньшим номером, он ждет своей очереди.
3. **Критическая секция**: Когда очередь доходит до процесса, он выполняет критическую секцию.
4. **Освобождение критической секции**: После выхода из критической секции процесс сбрасывает свой номер талончика в 0.

Этот алгоритм обеспечивает корректное взаимодействие нескольких процессов, удовлетворяя требованиям взаимного исключения, прогресса и ограниченного ожидания.