Теоретические вопросы к экзамену «Операционные системы»

1. Понятие операционной системы, её основные функции.
Операционная система (ОС) — это комплекс специального программного обеспечения, который управляет взаимодействием пользователей, прикладных программ с аппаратным обеспечением компьютера, обеспечивая эффективную, стабильную, защищенную работу всех функций, рационально распределяя и изолируя системные ресурсы. Иначе говоря, ОС предоставляет своего рода абстракцию над аппаратным обеспечением для упрощения разработки прикладных программ, осуществляя контроль и планирование использования таких ресурсов как процессорное время, память, устройства ввода-вывода, внешние накопители и т.д.

1. Управление процессами
2. Управление памятью
3. Управление файловой системой
4. Управление устройствами ввода-вывода
5. Обеспечение многозадачности
6. Обеспечение защиты и безопасности
7. Реализация пользовательского интерфейса
8. Поддержка сетевых функций
9. Работа с ошибками системы

2. Организация управления процессами в операционных системах.
Процесс в операционной системе Linux представляет собой выполняемую программу с набором ресурсов, таких как память, файловые дескрипторы и регистры процессора. Он выполняется изолированно от других процессов, что обеспечивает безопасность данных и устойчивость системы. Каждому процессу присваивается уникальный идентификатор PID, а также атрибуты, такие как родительский процесс (PPID) и приоритет. Создание нового процесса осуществляется через системный вызов fork(), который создаёт дочерний процесс, являющийся копией родительского. Родительский и дочерний процессы независимы, и их действия не влияют друг на друга. При необходимости выполнения другой программы дочерний процесс может использовать системные вызовы exec() для подмены кода.
(спрашивал у Островского как на 2 вопрос отвечать, сказал начинать с понятия процесс)

Планировщик — это компонент ядра операционной системы, который управляет распределением процессорного времени между различными процессами и потоками. Работает в режиме ядра и играет ключевую роль в многозадачности, обеспечивая эффективное выполнение всех процессов в системе.

Основные функции планировщика:
1) Планировщик распределяет процессорное время, то есть определяет, какой процесс или поток будет выполняться в данный момент, чтобы максимально эффективно использовать ресурсы процессора.
2) Планировщик учитывает приоритеты процессов, чтобы важные задачи, такие как системные или высокоприоритетные пользовательские процессы, могли получить доступ к процессору быстрее.
3) Планировщик управляет многозадачностью, то есть для поддержания одновременного выполнения нескольких задач он переключает процессы на выполнение, приостанавливая одни процессы и передавая процессорное время другим. Этот процесс называется контекстным переключением.
4) Планировщик выполняет оптимизацию производительности, то есть он равномерно распределяет ресурсы и предотвращает простаивания процессора, особенно если задачи связаны с интенсивными вычислениями или высокими требованиями к быстродействию.
5) Планировщик может применять временные ограничения на выполнение процессов, чтобы избегать монополизации ресурсов отдельными процессами, поддерживая стабильность и отзывчивость системы.

3. Механизмы управления памятью в операционных системах.
Виртуальная память — это технология управления оперативной памятью, позволяющая операционной системе исполнять программы, требующие больше памяти, чем физически доступно. Этот метод работает за счёт перемещения данных между основной памятью и вторичным хранилищем (жёстким диском или SSD), предоставляя программам непрерывное адресное пространство.
(спросил у Островского, сказал начать с определения виртуальной памяти)

Как работает виртуальная память:

1) Сегментация памяти (Memory Segmentation)

Когда приложение запускается, ему выделяется диапазон адресов виртуальной памяти. Эта память разделена на блоки (chunks), называемые страницами.

2) Таблица страниц (Page Tables)

Операционная система сохраняет для каждого процесса структуру данных, которая называется таблицей страниц. Таблица страниц сопоставляет адреса страниц виртуальной памяти с адресами страниц физической памяти.

3) Доступ к памяти (Memory Access)

Когда приложение выполняет запрос на чтение или запись к памяти, ЦПУ использует таблицу страниц для трансляции адреса виртуальной памяти в адрес физической.

4) Подкачка и запрос страниц (Swapping and Demand Paging)

Если вся физическая память использована, а приложению требуется загрузить новую страницу в нее, то ОС может выбрать страницу физической памяти для

сброса-«подкачки» (swap out) на диск. В таблицу страниц вносится пометка, что данная страница отсутствует в физической памяти. Если позднее приложение попытается получить доступ к адресу выгруженной на диск страницы, то это приведет к ошибке, и операционная система загрузит-«подкачает» (swap in) страницу с диска в физическую память (возможно сбросив при этом на диск другую страницу), после чего приложение сможет получить к ней доступ.

4. Виртуальная память.
Виртуальная память — это технология управления оперативной памятью, позволяющая операционной системе исполнять программы, требующие больше памяти, чем физически доступно. Этот метод работает за счёт перемещения данных между основной памятью и вторичным хранилищем (жёстким диском или SSD), предоставляя программам непрерывное адресное пространство.

В монолитных ядрах операционных систем виртуальные адреса преобразуются в физические с помощью специального аппаратного модуля — блока управления памятью (MMU), который управляет таблицами страниц и буфером ассоциативной трансляции (TLB).

5. Архитектуры ядер операционных систем (обзор).

- Монолитное ядро — это классическая архитектура операционных систем, в которой все функции ядра работают в общем адресном пространстве, обеспечивая высокую производительность и богатый набор абстракций оборудования. Хотя монолитное ядро является зрелой и надежной архитектурой, его сложная структура затрудняет отладку и расширение, повышает требования к памяти и усложняет использование в системах с ограниченными ресурсами, например, во встраиваемых системах
- Микроядро — это тип ядра операционной системы, которое реализует только минимальный набор функций, необходимых для базовой работы системы. В отличие от монолитного ядра, микроядро включает лишь основные службы, такие как управление памятью, процессами, вводом-выводом, а также межпроцессное взаимодействие (IPC). Вся остальная функциональность, включая файловые системы и драйверы устройств, выполняется в виде отдельных процессов в пользовательском пространстве.
- Гибридное ядро — это тип архитектуры операционной системы, сочетающий элементы микроядерной и монолитной архитектур. В отличие от чистого микроядра, гибридное ядро допускает размещение некоторых модулей ОС в пространстве ядра для повышения производительности, обеспечивая при этом структурированное разделение компонентов. Основной идеей гибридного ядра является балансировка между безопасностью и стабильностью микроядра и производительностью монолитного ядра.

6. Особенности монолитного ядра операционной системы: преимущества и недостатки.
Монолитное ядро — это классическая архитектура операционных систем, в которой все функции ядра работают в общем адресном пространстве, обеспечивая высокую производительность и богатый набор абстракций

оборудования. Хотя монолитное ядро является зрелой и надежной архитектурой, его сложная структура затрудняет отладку и расширение, повышает требования к памяти и усложняет использование в системах с ограниченными ресурсами, например, во встраиваемых системах. поскольку всё ядро работает в одном адресном пространстве, сбой в одном из компонентов может нарушить работоспособность всей системы.

7. Микроядро операционной системы.
Микроядро — это тип ядра операционной системы, которое реализует только минимальный набор функций, необходимых для базовой работы системы. В отличие от монолитного ядра, микроядро включает лишь основные службы, такие как управление памятью, процессами, вводом-выводом, а также межпроцессное взаимодействие (IPC). Вся остальная функциональность, включая файловые системы и драйверы устройств, выполняется в виде отдельных процессов в пользовательском пространстве.
(определение микроядра)

Некоторые известные микроядерные операционные системы:

- MINIX. Операционная система, изначально разработанная в качестве образовательного проекта. Отличается небольшим размером и упрощенной структурой.
- QNX. Коммерческая микроядерная ОС, предназначенная для встраиваемых систем. Поддерживает многозадачность, обладает повышенной безопасностью и отказоустойчивостью.
- L4. Семейство микроядер, разработанное для встраиваемых систем и мобильных устройств. Микроядра L4 отличаются высокой производительностью, низкой задержкой, безопасностью и универсальностью.
- Hurd. Операционная система, разработанная как альтернатива ядру Linux с упором на модульность и масштабируемость.
- PikeOS. Микроядерная операционная система реального времени (RTOS), разработанная для встраиваемых систем с высокими требованиями к надёжности и безопасности.

(ответ нейронки)

8. Гибридное ядро: концепция и примеры операционных систем, использующих этот подход.
Это тип архитектуры операционной системы, сочетающий элементы микроядерной и монолитной архитектур. В отличие от чистого микроядра, гибридное ядро допускает размещение некоторых модулей ОС в пространстве ядра для повышения производительности, обеспечивая при этом структурированное разделение компонентов.
К гибридным ядрам относят. MacOS X, Windows NT, DragonFly BSD, поскольку значительная часть системных служб и драйверов данных систем реализуется в виде процессов пользовательского режима, то есть активно используется архитектура микроядра.

9. Многозадачность. Поддержка многозадачности в операционных системах.

ОС обеспечивает многозадачность, позволяя нескольким программам выполняться одновременно, и многопользовательский режим, раздельно управляя доступом к данным и ресурсам для каждого пользователя

10. Виртуальная файловая система (VFS).
Виртуальная файловая система (VFS) — это уровень абстракции в операционной системе, позволяющий унифицировать доступ к различным файловым системам через единый интерфейс. VFS обеспечивает работу с разными файловыми системами (например, ext4, NFS, FAT32) как с локальными, так и с сетевыми устройствами, а также поддерживает специальные файловые системы, такие как procfs. Главная задача VFS — предоставить стандартный интерфейс для взаимодействия между ядром операционной системы и файловыми системами.

11. Межпроцессное взаимодействие (IPC) в операционной системе Linux.
Межпроцессное взаимодействие (IPC) — это система обмена данными и координации работы между потоками или процессами в операционной системе, необходимая для синхронизации, обмена данными и распределения ресурсов. В операционных системах с монолитным ядром IPC реализует множество механизмов, таких как файлы, сигналы, каналы, семафоры, очереди сообщений и сокеты, обеспечивая гибкость для различных задач и взаимодействий.

12. Управление устройствами ввода-вывода в операционных системах.
ОС координирует взаимодействие с периферийными устройствами (например, клавиатурой, принтером, дисками), обеспечивая стандартизированный интерфейс для программ.

13. Эволюция операционных систем: основные этапы развития.
1. Пакетная обработка (1950-е — 1960-е гг.). Первые ОС были ориентированы на пакетную обработку данных, что позволило пользователям готовить задания на перфокартах и загружать их в систему. ОС управляла очередью задач, минимизируя время простоя оборудования.

2. Временное разделение (1960-е — 1970-е гг.) С появлением концепции времени разделения (timesharing) пользователи могли одновременно работать на одном компьютере. ОС начала поддерживать многозадачность, распределяя время процессора между различными задачами.

3. Персональные компьютеры (1980-е гг.) Появление персонального компьютера изменило парадигму ОС. Появились такие ОС, как MS-DOS и Windows, ориентированные на одиночного пользователя и взаимодействие с пользователем через графический интерфейс.

4. Сетевые и распределенные системы (1990-е гг.). С развитием компьютерных сетей возникла необходимость в ОС, поддерживающих сетевые функции и распределённые вычисления. ОС, такие как UNIX и Windows NT, начали интегрировать сетевые функции и поддержку многоядерных процессоров.

5. Мобильные и встраиваемые системы (2000-е — настоящее время) Появление смартфонов и планшетов вызвало необходимость создания специализированных ОС, таких как iOS и Android, ориентированных на мобильные устройства и сенсорные интерфейсы.

14. Стандарты POSIX, их влияние на развитие операционных систем.
Стандарты POSIX (Portable Operating System Interface) — это набор стандартов, разработанных для обеспечения совместимости между операционными системами, особенно в контексте UNIX-подобных систем. Эти стандарты определяют интерфейсы программирования, командные оболочки и утилиты, которые должны поддерживаться операционными системами для обеспечения переносимости приложений.

Основные аспекты стандартов POSIX:

1. Совместимость:
    - POSIX обеспечивает стандартные интерфейсы для системного программирования, что позволяет разработчикам писать переносимые приложения, которые могут работать на различных UNIX-подобных операционных системах без необходимости изменения кода.
2. Определение API:
    - Стандарты определяют набор системных вызовов и библиотек, которые должны поддерживаться операционными системами. Это включает в себя управление процессами, управление памятью, файловые операции и сетевые функции.
3. Командные оболочки и утилиты:
    - POSIX включает спецификации для командных оболочек и утилит, что обеспечивает единообразие в командной строке и утилитах для управления системой.

Влияние POSIX на развитие операционных систем:

1. Упрощение разработки:
    - Стандарты POSIX значительно упростили процесс разработки программного обеспечения, так как разработчики могут использовать одни и те же API и утилиты на различных системах, что снижает затраты на разработку и тестирование.
2. Увеличение совместимости:

    - POSIX способствовал повышению совместимости между различными UNIX-подобными операционными системами, такими как Linux, FreeBSD, Solaris и другими. Это позволило пользователям и разработчикам легче переходить между системами.
3. Расширение экосистемы:
    - Благодаря стандартам POSIX, разработчики могут создавать приложения, которые работают на различных платформах, что способствовало расширению экосистемы программного обеспечения для UNIX-подобных систем.
4. Влияние на другие операционные системы:
    - POSIX оказал влияние не только на UNIX-подобные системы, но и на другие операционные системы, такие как Windows, которая внедрила поддержку POSIX через различные интерфейсы и подсистемы.

15. Управление файловой системой: создание, удаление и доступ к файлам.

Создание пустого файла: touch file.txt
Создание нескольких файлов: touch file1.txt file2.txt file3.txt
Удаление файлов с помощью команды rm (например, rm file.txt) и директорий с помощью rm -r (например, rm -r directory_name).
Просмотр содержимого текстовых файлов с помощью команды cat (например, cat file.txt).

16. Драйвер, его взаимодействие с операционной системой.

Драйверы в операционной системе Linux являются ключевыми компонентами, обеспечивающими взаимодействие операционной системы с аппаратным обеспечением. Они предоставляют стандартизированный интерфейс, абстрагируя особенности реализации конкретных устройств и упрощая их использование. Основная функция драйвера заключается в предоставлении операционной системе возможности работать с устройствами без необходимости знания их внутреннего устройства.
С точки зрения архитектуры, драйверы работают на уровне ядра и используют системные вызовы для выполнения своих функций. Они могут предоставлять API через интерфейсы файловой системы, такие как /dev, обеспечивая доступ через стандартные операции read(), write(), open() и close(). Это делает управление устройствами прозрачным для приложений, которые обращаются к ним как к обычным файлам.

17. Обеспечение защиты и безопасности в современных операционных системах.

ОС обеспечивает защиту данных и ресурсов, предотвращая несанкционированный доступ и защищая систему от потенциальных угроз. Управляет правами доступа, обеспечивает аутентификацию пользователей.

18. Планировщик задач в операционной системе. Основные алгоритмы.
Планировщик задач в Linux управляет распределением процессорного времени между процессами и потоками, обеспечивая эффективное выполнение задач. Его задачи включают максимизацию пропускной способности и минимизацию времени ожидания и отклика. В Linux процессы делятся на два типа: процессы реального времени и условные процессы. Для процессов реального времени используются политики SCHED_FIFO и SCHED_RR. Политика SCHED_FIFO даёт приоритет первым поступившим процессам, а SCHED_RR реализует циклическое планирование с фиксированными квантами времени. Условные процессы управляются с помощью Completely Fair Scheduler (CFS), который равномерно распределяет процессорное время

19. Системные вызовы в операционной системе (на примере ОС Linux).
Системный вызов — это механизм, через который прикладные программы обращаются к ядру операционной системы для выполнения операций, требующих привилегированного доступа к ресурсам, таких как управление памятью, работа с файлами и доступ к устройствам ввода-вывода. При системном вызове приложение передает управление ядру, которое проверяет права доступа и выполняет операцию от имени приложения. Этот механизм позволяет пользователям взаимодействовать с аппаратными ресурсами через интерфейс ОС, при этом программы работают в ограниченном режиме, а ядро — в привилегированном режиме

20. Разделяемая память в операционной системе.
Разделяемая память — один из самых быстрых механизмов IPC, позволяющий нескольким процессам одновременно работать с одним участком памяти. Чтобы избежать конфликтов при одновременном доступе, разделяемая память часто используется вместе с семафорами, которые синхронизируют доступ к ресурсам, предотвращая ситуации гонки.

Очереди сообщений позволяют процессам обмениваться структурированными сообщениями, поддерживая асинхронный обмен, что облегчает взаимодействие без необходимости синхронизации по времени.

21. Алгоритмы подкачки (swapping) в виртуальной памяти.
Виртуальная память в большинстве операционных систем организована страничным способом, при котором память разделяется на блоки фиксированного размера — страницы. Эта технология поддерживает совместное использование одних и тех же данных между процессами, что позволяет эффективно использовать ресурсы системы.

Алгоритм подкачки полностью перемещает давно не использованные ресурсы выделенные на процесс в постоянное хранилище (не использует страницы).

22. Сравнение ОС Windows, Linux и macOS (обзор).
Windows более закрытая система, многие операции невозможно сделать через консоль (только через графический интерфейс). Больше направлена на пользователей далеких от знаний компьютера.

Linux невероятно гибкая система; имеет открытый исходный код, поэтому имеет множество дистрибутивов. Подавляющее большинство действий можно произвести через команды. Используется для организации серверов, обеспечивает безопасность на низком уровне.

MacOS базируется на Unix-системе (как и Linux), имеет небольшое кол-во отличий по сравнению с Linux; глубоко интегрирован в экосистему Apple.

23. Поддержка сетевых функций в операционной системе Linux. – можно дополнить
Linux предоставляет мощные инструменты для работы с сетью, начиная от базовой настройки интерфейсов и заканчивая сложными сценариями маршрутизации, виртуализации и мониторинга.

Сетевые функции делятся:
1. Сетевые протоколы
2. Сетевые интерфейсы
3. Маршрутизация
4. Брандмауэр и фильтрация трафика
5. Виртуализация и контейнеризация
6. Сетевые утилиты
7. Сетевые сервисы
8. Сетевые файловые системы
9. Настройка сети через конфигурационные файлы
10. Поддержка Wi-Fi
11. Сетевые пространства имён (Network Namespaces)
12. Поддержка SDN (Software-Defined Networking)
13. Мониторинг сети
14. Поддержка высоконагруженных сетей

/etc/sysconfig/network-scripts/ - путь для конфигурации сетевых интерфейсов.

Некоторые команды:

scp - утилиту для копирования по ssh (SSH — сетевой протокол прикладного уровня, предназначенный для безопасного удаленного доступа к различным системам),
curl - утилита для запросов,
ping - для проверки коннектов,
ipconfig - для проверки текущих соединений

24. Потоки. Управление потоками в операционной системе Linux.
Поток — это основная единица активности внутри процесса, выполняющая код и поддерживающая процесс в рабочем состоянии. Процессы обычно бывают либо однопоточными, либо многопоточными, причем большинство процессов — однопоточные.

В системах UNIX процессы традиционно однопоточны из-за высокой скорости создания процессов и надежных методов межпроцессного взаимодействия. Однако Linux поддерживает потоки на уровне ядра, где потоки по своей сути представляют собой процессы с общими ресурсами.

25. Особенности ОС, разработанных для мобильных и встроенных систем.
Для таких платформ используется не все модули ядра linux, некоторые могут исключены или замены, это достигается за счет микросервисной архитектуры и открытости ядра Линукс.

26. Механизмы изоляции процессов в современных операционных системах.
IPC обеспечивает изоляцию процессов, позволяя каждому работать в своём адресном пространстве и защищая ресурсы от несанкционированного доступа.

Виртуальная память также обеспечивает изоляцию процессов, предотвращая доступ к памяти одного процесса со стороны другого. Это достигается благодаря независимым виртуальным адресным пространствам и аппаратной поддержке защиты памяти.

Для обеспечения изоляции процессов и предоставления каждому процессу независимого виртуального адресного пространства используются комплексные алгоритмы и механизмы, направленные на предотвращение несанкционированного доступа к памяти между процессами.

Основу изоляции составляют алгоритмы виртуальной памяти, реализующие перевод виртуальных адресов в физические с помощью таблиц страниц (Page Tables). Этот механизм позволяет каждому процессу работать с собственным виртуальным адресным пространством, независимым от других, и исключает

пересечения благодаря точному отображению виртуальных адресов в физическую память.

Важным компонентом изоляции являются контекстные переключения. При переключении между процессами операционная система обновляет данные о виртуальном адресном пространстве, используя регистры базового адреса таблицы страниц. Это гарантирует, что процессы не могут обращаться к памяти друг друга

27. Кибербезопасность в контексте операционных систем.
Преимущества Линукс состоят в:

- Открытый исходный код: Позволяет сообществу находить и исправлять уязвимости.
- Модульность: Linux состоит из множества независимых компонентов, что снижает риск распространения атак. И позволяет подключать шифраторы и настраивать межсетевые экраны на различных уровнях системы.
- Минимальные привилегии: Принцип "наименьших привилегий" (Least Privilege) ограничивает доступ пользователей и процессов к критическим ресурсам.
- Разделение прав: Чёткое разделение прав между пользователем и администратором (root).

28. Файловые системы в управлении данными и доступом к ресурсам.
Файловые системы в таких ОС организованы иерархически с единой корневой директорией, в которой все устройства и каталоги включены в общую структуру.

Файловые системы могут включать разнообразные функции, такие как сжатие данных, контроль доступа, шифрование и управление атрибутами файлов. Специальные файловые системы, например, procfs, предоставляют доступ к информации об аппаратных и системных ресурсах, выполняя роль вспомогательных файловых структур для работы с параметрами ядра и диагностики.

Права доступа выставляются 3 битами значащими доступ на чтение (r), запись (w), выполнение (x) соответственно и могут определяться для владельца, группы и остальных пользователей. Пример записи rwxr-xr-- (владелец может читать, писать и выполнять; группа — читать и выполнять; остальные — только читать). Пример команд: chmod 755 file.txt ; chown username:groupname file.txt

29. Операционная система Linux (обзор).
Основные особенности:

- Открытый исходный код: Linux распространяется под лицензией GPL (GNU General Public License), что позволяет свободно использовать, модифицировать и распространять систему.
- Многозадачность и многопользовательский режим: Linux поддерживает одновременную работу множества пользователей и процессов.
- Стабильность и надежность: Linux известен своей устойчивостью к сбоям и способностью работать без перезагрузки в течение длительного времени.
- Гибкость: Linux может быть настроен для выполнения самых разных задач, от простого настольного компьютера до мощного сервера.
- Безопасность: Благодаря открытому исходному коду и активному сообществу, уязвимости быстро обнаруживаются и исправляются.

30. Процессы. Базовая работа с процессами в операционной системе Linux.

Процесс в операционной системе Linux представляет собой выполняемую программу с набором ресурсов, таких как память, файловые дескрипторы и регистры процессора. Он выполняется изолированно от других процессов, что обеспечивает безопасность данных и устойчивость системы. Каждому процессу присваивается уникальный идентификатор PID, а также атрибуты, такие как родительский процесс (PPID) и приоритет. Создание нового процесса осуществляется через системный вызов fork(), который создаёт дочерний процесс, являющийся копией родительского. Родительский и дочерний процессы независимы, и их действия не влияют друг на друга. При необходимости выполнения другой программы дочерний процесс может использовать системные вызовы exec() для подмены кода.

31. Системный вызов fork() в операционной системе Linux. Примеры.

Каждому процессу присваивается уникальный идентификатор PID, а также атрибуты, такие как родительский процесс (PPID) и приоритет. Создание нового процесса осуществляется через системный вызов fork(), который создаёт дочерний процесс, являющийся копией родительского. Родительский и дочерний процессы независимы, и их действия не влияют друг на друга.

В обоих процессах выполнение продолжается с той же строки, где был вызван fork(). В родительском процессе fork() возвращает PID дочернего процесса, что позволяет идентифицировать его, а в дочернем процессе — 0.

```
#include
#include
int main() {
	pid_t pid = fork();
	if (pid < 0) {
```

```
        // Ошибка при создании дочернего процесса
        perror("Ошибка при создании дочернего процесса");
        return -1;
} else if (pid == 0) {
        // Код дочернего процесса
        printf("Родитель %d => %d Потомок\n", getppid(), getpid());
} else {
        // Код родительского процесса
        printf("Корневой процесс %d\n", getpid()); }
printf("Это сообщение печатает процесс: %d\n", getpid());
return 0; }
```

32. Жизненный цикл процесса в операционной системе Linux.

Рис. 4. Жизненный цикл процесса в ОС Linux

Жизненный цикл процесса представляет собой цепочку состояний, таких как создание, готовность, выполнение, ожидание и завершение, с возможностью временной приостановки и возврата к готовности, пока процесс окончательно не завершит свою работу. (больше инфы можно узнать на страницах 26-27 пособия)

33. Управление процессами без использования парадигм межпроцессного взаимодействия.

Системный вызов ptrace() в Linux используется для отладки процессов. Он позволяет одному процессу контролировать выполнение другого процесса.

Программа демонстрирует управление одним процессом (родителем) другим процессом (потомком) с использованием системного вызова ptrace(). Сначала процесс разделяется с помощью fork(), и создается потомок. Потомок начинает выполнять вычисление суммы натуральных чисел, но прерывается досрочно на

третьей итерации. Он сообщает родителю о намерении отслеживаться, используя PTRACE_TRACEME, а затем сохраняет промежуточные результаты в регистрах процессора и останавливает свое выполнение с помощью SIGSTOP. Родитель ожидает остановки потомка и использует ptrace с командой PTRACE_GETREGS для доступа к значениям регистров потомка, где хранятся данные о промежуточных вычислениях. Получив эти данные, родитель продолжает вычисление суммы, начиная с той итерации, на которой остановился потомок, и завершает его. После завершения вычислений родитель печатает полученный результат, а затем возобновляет выполнение потомка, используя PTRACE_CONT. Родитель снова ожидает завершения потомка, и программа завершается (программа находится на странице 29)

34. Парадигмы межпроцессного взаимодействия. Разделяемая память.
Разделяемая память (sys/shm.h)
Разделяемая память в Linux — это механизм межпроцессной коммуникации, который позволяет нескольким процессам получать доступ и манипулировать общим блоком памяти. Он обеспечивает эффективный обмен данными и синхронизацию между процессами без необходимости в сложных методах передачи данных, таких как передача сообщений или ввод-вывод файлов.

С помощью системного вызова fork() создается новый процесс, который наследует все ресурсы родительского процесса, включая разделяемую память. Такая память становится доступной и для родительского, и для дочернего процесса, что удобно для межпроцессного взаимодействия. В UNIX существуют два основных подхода к созданию и использованию совместно используемой памяти. Первый подход использует стиль UNIX System V и включает функции shmget, shmctl, shmat и shmdt. Функция shmget создает сегмент совместно используемой памяти, shmctl позволяет управлять его параметрами, а shmat и shmdt присоединяют и отключают сегмент от адресного пространства процесса. Второй способ, основанный на POSIX Shared Memory, применяет функции shm_open, shm_unlink, ftruncate и mmap. В отличие от UNIX System V, POSIX позволяет связать объект совместно используемой памяти с файловым дескриптором, что упрощает управление памятью. Функция shm_open создает или подключает объект совместно используемой памяти по имени, а shm_unlink удаляет его. Функция ftruncate задает размер памяти, а mmap отображает ее в адресное пространство процесса. Совместно используемая память в POSIX стандартизирована и может быть легко подключена через имя объекта, что делает этот метод удобным и гибким.

35. Механизмы синхронизации в операционной системе Linux. Семафоры.
Семафоры в POSIX — это механизм синхронизации, который используется для управления доступом к общим ресурсам в многозадачных средах. POSIX определяет тип данных семафора sem_t и функции для его инициализации, управления и освобождения. Функция sem_init() создаёт семафор, задавая начальное значение счётчика, и может использоваться для синхронизации на уровне потоков и процессов. Семафоры POSIX позволяют создавать именованные и неименованные объекты. Именованные семафоры создаются с помощью sem_open(), а sem_unlink() удаляет их имя, освобождая память. Функция sem_wait() уменьшает значение семафора на единицу, блокируя выполнение, если ресурс недоступен. sem_post(), напротив, увеличивает значение семафора, сигнализируя об освобождении ресурса.

36. Механизмы синхронизации в операционной системе Linux. Мьютексы.
Мьютексы (от английского mutual exclusion) — это ключевые механизмы синхронизации потоков, предназначенные для обеспечения взаимного исключения при доступе к разделяемым ресурсам. Они позволяют

гарантировать, что только один поток в определённый момент времени может работать с критической секцией, где изменяются или используются общие данные.
В POSIX API мьютексы представлены типом данных pthread_mutex_t, который предоставляет разработчикам инструменты для их создания и управления. Для инициализации мьютекса используется функция pthread_mutex_init(), принимающая указатель на объект мьютекса и атрибуты. Если настройки не требуются, можно передать значение NULL, и мьютекс будет создан с параметрами по умолчанию.
Для обеспечения взаимного исключения используется механизм мьютексов с функциями pthread_mutex_lock() и pthread_mutex_unlock().

37. Механизмы синхронизации в операционной системе Linux. Механизм блокировки файлов.

Механизм блокировки файлов (flock) Механизм блокировки файлов в POSIX-системах, включая Linux, используется для управления доступом к файлам с целью предотвращения конфликта операций, например, одновременной записи и чтения. Основной функцией блокировок является предотвращение состояний гонки, когда одновременное выполнение операций приводит к потерям данных или некорректным результатам.

Системный вызов flock() предоставляет интерфейс для установки блокировок на файлы: разделяемых (shared) и эксклюзивных (exclusive). Разделяемая блокировка (shared lock) позволяет нескольким процессам одновременно читать файл, но запрещает запись.

Эксклюзивная блокировка (exclusive lock) запрещает другим процессам доступ к файлу до снятия блокировки. Да, чтобы использовать эксклюзивную блокировку, нужно в приведенном примере заменить параметр LOCK_SH (разделяемая блокировка) на LOCK_EX (эксклюзивная блокировка).

38. Механизмы синхронизации в операционной системе Linux. Именованный канал FIFO.

Именованный канал FIFO (First In, First Out) в Linux представляет собой механизм межпроцессного взаимодействия (IPC), позволяющий обмениваться данными между процессами, даже если они не имеют родственной связи. FIFO является специальным файлом в файловой системе, создаваемым с помощью системного вызова mkfifo(), который возвращает 0 при успешном выполнении или -1 при ошибке. Этот файл функционирует как труба, позволяя процессам записывать данные с одной стороны и читать с другой, сохраняя порядок "первым пришел - первым ушел".

Создание FIFO сопровождается указанием имени файла и прав доступа. При записи в FIFO важно учитывать ограничение PIPE_BUF для предотвращения

смешивания данных от нескольких процессов. Если канал не открыт для чтения, попытка записи вызывает сигнал SIGPIPE, который можно обработать для корректного завершения программы.

39. Механизмы синхронизации в операционной системе Linux. Очереди сообщений.

Очереди сообщений (Message Queues) — это один из механизмов межпроцессного взаимодействия (IPC) в Linux, который позволяет процессам обмениваться структурированными данными в асинхронном режиме. Основное преимущество очередей сообщений заключается в их способности поддерживать приоритет сообщений, что позволяет процессам выбирать порядок обработки данных. Каждый процесс может добавить сообщение в очередь с помощью вызова msgsnd или извлечь сообщение через msgrcv.

Команда ipcs позволяет просматривать существующие очереди, их параметры, такие как размер, количество сообщений и права доступа.

Очереди сообщений создаются с помощью вызова msgget, который принимает ключ и флаги, например IPC_CREAT, создающий очередь, если она еще не существует. Права доступа на очередь задаются в восьмеричном формате, аналогично файловым системам Linux. Каждое сообщение, добавленное в очередь, содержит два основных поля: mtype (тип сообщения) и mtext (содержимое). Тип сообщения позволяет организовать фильтрацию, а содержимое может быть текстом или любыми другими данными.

Функция msgsnd отправляет сообщение в очередь. Если очередь заполнена, поведение определяется флагом msgflg. Если установлен IPC_NOWAIT, вызов завершится с ошибкой, иначе процесс будет заблокирован до появления свободного места. Функция msgrcv извлекает сообщение из очереди, также поддерживая фильтрацию по типу сообщения, что позволяет выбирать сообщения с заданным mtype или получать сообщения в порядке их добавления.

40. Механизмы синхронизации в операционной системе Linux. Локальные сокеты.

Локальные сокеты (UNIX Domain Sockets) представляют собой механизм межпроцессного взаимодействия (IPC) в ОС Linux, основанный на API сокетов Беркли. Они используют адресацию через файловую систему и обеспечивают высокую производительность за счёт работы внутри одной системы без сетевых накладных расходов. UNIX-сокеты, определённые в , функционируют через файловые дескрипторы и позволяют передавать данные между процессами через файловые пути. Создание сокета осуществляется вызовом socket() с параметром PF_UNIX, после чего следует привязка к файлу с использованием bind(). Для отправки и получения данных применяются функции send() и recv(), аналогичные TCP/UDP сокетам. UNIX-сокеты бывают потоковыми (SOCK_STREAM) и датаграммными (SOCK_DGRAM), обеспечивая надежную и быструю передачу данных или обмен сообщениями без установки соединений.

41. Механизм Memory-Mapped Files.
Memory-Mapped Files (mmap) — это системный вызов в Unix-подобных системах, который позволяет отобразить файл или устройство на виртуальное адресное пространство процесса. Этот механизм обеспечивает эффективный способ ввода-вывода, так как данные не сразу загружаются в память, а считываются в «ленивом» режиме, только при обращении к определённой области.

Одним из ключевых преимуществ mmap является возможность организации межпроцессного взаимодействия. Если отображение памяти выполняется с флагом MAP_SHARED, то изменения, выполненные одним процессом, становятся видимыми для других процессов, использующих то же отображение. Это делает mmap мощным инструментом для организации совместного доступа к данным.

42. Парадигмы межпроцессного взаимодействия. Механизм эвентов.
Механизм eventfd предоставляет простой и эффективный способ межпроцессного взаимодействия (IPC) в Linux, позволяя уведомлять процессы о событиях. Системный вызов eventfd() создает файловый дескриптор, связанный с 64-битным счетчиком, который поддерживается ядром и инициализируется значением, указанным пользователем. Этот файловый дескриптор используется для чтения и записи событий, позволяя процессам уведомлять друг друга. Счетчик может быть увеличен с помощью операции записи write() и уменьшен или сброшен до нуля с помощью чтения read(). Флаг EFD_SEMAPHORE позволяет использовать eventfd в семафоро-подобной семантике, уменьшая счетчик на единицу при каждом чтении. В отличие от каналов (pipe), eventfd требует меньших накладных расходов и использует один файловый дескриптор вместо двух.

Флаг EFD_NONBLOCK позволяет избежать блокировок при чтении или записи, что полезно в асинхронных приложениях. Флаг EFD_CLOEXEC автоматически закрывает файловый дескриптор при вызове execve(). Механизм поддерживает интерфейсы мультиплексирования, такие как poll, select и epoll, позволяя эффективно отслеживать готовность к чтению или записи наряду с другими файловыми дескрипторами. Значение счетчика может быть просмотрено через файловую систему /proc

Пример демонстрирует межпроцессное взаимодействие с использованием механизма eventfd. Сначала создается файловый дескриптор efd, связанный с 64-битным счетчиком, который инициализируется значением 0. После вызова fork() создаются два процесса: родительский и дочерний. Дочерний процесс записывает значение 5 в eventfd, увеличивая счетчик, а родительский процесс читает это значение, одновременно сбрасывая счетчик до 0. Таким образом, родительский процесс успешно получает уведомление о событии,

инициированном дочерним процессом, демонстрируя простой и эффективный способ синхронизации.

43. Парадигмы межпроцессного взаимодействия. Анонимные каналы.
Неименованные каналы (анонимные каналы, pipe) — это одно из основных средств межпроцессного взаимодействия (IPC) в операционных системах, таких как Linux. Они обеспечивают однонаправленный обмен данными между родительским и дочерним процессами, используя стратегию FIFO (первый записан — первый прочитан). Для их создания используется системный вызов pipe(), возвращающий два файловых дескриптора: один для записи, другой для чтения. После выполнения fork(), дочерний процесс наследует дескрипторы родительского процесса, что позволяет организовать передачу данных. Если канал используется только для записи или чтения, избыточные дескрипторы закрываются. При отсутствии данных read() блокирует процесс до появления информации, а при полном канале write() — до освобождения места. Чтобы избежать блокировок, дескрипторы могут быть переведены в неблокирующий режим с помощью fcntl(). Размер буфера канала ограничен (например, 4096 байт в Linux), и записи объемом, превышающим PIPE_BUF, могут быть неатомарными. При закрытии одного конца канала read() возвращает 0, а write() вызывает сигнал SIGPIPE, сигнализирующий о невозможности записи. Основное преимущество pipe — высокая скорость, так как данные не записываются в файловую систему, а хранятся в памяти ядра. Однако анонимные каналы ограничиваются родственными процессами, функционируют только на одном компьютере и не сохраняют границ сообщений.

44. Сигналы в в операционной системе Linux (обзор). Примеры использования.
Сигналы в контексте межпроцессного взаимодействия (IPC) в операционной системе Linux представляют собой асинхронный механизм уведомления процессов о наступлении событий. Сигналы могут быть отправлены ядром, другими процессами или самим процессом, обеспечивая управление событиями, такими как прерывание, завершение, приостановка или возобновление процесса. Они являются основным средством межпроцессного взаимодействия в Unix-подобных системах и предоставляют гибкие возможности обработки. Каждый сигнал имеет уникальное символьное название (например, SIGINT, SIGTERM) и числовое значение, определяемое в заголовочном файле signal.h. Большинство сигналов имеют стандартное поведение по умолчанию, которое может быть изменено процессом с использованием пользовательских обработчиков.

Асинхронная природа сигналов позволяет отправлять их в любой момент времени, независимо от текущего состояния процесса. Процессы могут самостоятельно назначать обработчики сигналов, за исключением двух: SIGKILL (безусловное завершение) и SIGSTOP (приостановка), которые невозможно перехватить или игнорировать. Для отправки сигналов используется системный вызов kill(), который передаёт сигнал процессу по его идентификатору (PID).

45. Механизмы синхронизации в операционной системе Linux. Условные переменные.

Условные переменные являются механизмом синхронизации в POSIX threads (pthread), предназначенным для управления ожиданием событий между потоками. Они используются совместно с мьютексами для безопасного доступа к общим данным и ожидания определенных условий.

Условная переменная связана с логическим выражением (предикатом), которое определяет текущее состояние данных. Если предикат ложен, поток может заснуть на условной переменной, ожидая изменения состояния. Поток пробуждается, когда другой поток сигнализирует об изменении предиката.

Для работы с условными переменными предусмотрены функции pthread_cond_wait, pthread_cond_signal и pthread_cond_broadcast. Функция pthread_cond_wait используется для ожидания, она принимает два аргумента: условную переменную и мьютекс. При вызове эта функция блокирует поток, освобождая мьютекс, что позволяет другим потокам изменить состояние предиката. После пробуждения поток снова захватывает мьютекс. Функция pthread_cond_signal пробуждает один из потоков, ожидающих на условной переменной. Если требуется разбудить все потоки, используется pthread_cond_broadcast. Использование условных переменных особенно эффективно в случаях, когда ожидание события требует синхронизации между несколькими потоками.

46. Механизмы синхронизации в операционной системе Linux. Фьютексы.

Фьютексы (fast user-space mutexes) — это низкоуровневый примитив синхронизации, разработанный для оптимизации взаимодействия потоков или процессов в операционной системе Linux.

Фьютекс представляет собой 4-байтное целое число, размещённое в общей памяти, доступной нескольким потокам или процессам. Его можно использовать для создания высокоуровневых абстракций синхронизации, таких как мьютексы, условные переменные и семафоры. В большинстве случаев фьютексы работают исключительно в пользовательском пространстве, и ядро привлекается только при возникновении конфликта.

Работа с фьютексом включает две основные операции: ожидание (FUTEX_WAIT) и пробуждение (FUTEX_WAKE). При бесконфликтном выполнении операции FUTEX_WAIT поток проверяет состояние фьютекса и при необходимости приостанавливается до наступления нужного события. FUTEX_WAKE, напротив, пробуждает ожидающие потоки, когда ресурс

становится доступным. Ключевым преимуществом фьютексов является их эффективность при работе в бесконфликтных условиях.

47. Механизмы синхронизации в операционной системе Linux. Барьерные синхронизации.

Барьерная синхронизация (pthread_barrier_t) – это механизм в POSIX для синхронизации выполнения потоков. Барьеры позволяют приостановить выполнение потоков до определённого момента, пока все они не достигнут заданной точки программы, после чего выполнение продолжается синхронно. Такой подход удобен для параллельной обработки данных, где потоки должны дождаться завершения задач друг друга перед началом нового этапа работы

Для создания барьера используется структура pthread_barrier_t. Инициализация выполняется с помощью функции pthread_barrier_init(), которая принимает три параметра: указатель на барьер, указатель на атрибуты и количество потоков, которые должны достигнуть барьера для его разблокировки. Если атрибуты не требуются, передаётся NULL. Число потоков должно быть больше нуля, иначе функция вернёт ошибку EINVAL.

После использования барьер необходимо уничтожить, вызвав функцию pthread_barrier_destroy(). Уничтожение барьера, который всё ещё используется потоками, может привести к ошибке EBUSY, поэтому уничтожение нужно выполнять только после завершения всех потоков, использующих этот барьер.

Синхронизация потоков через барьер осуществляется вызовом функции pthread_barrier_wait(). Поток, достигший этой функции, блокируется, пока не будет достигнуто заданное количество потоков. Как только все потоки достигают барьера, выполнение продолжается синхронно. Одному из потоков, преодолевших барьер, функция возвращает специальное значение PTHREAD_BARRIER_SERIAL_THREAD, а остальные потоки получают 0.

48. Механизмы синхронизации в операционной системе Linux. Спинлоки.

Спинлоки (pthread_spinlock_t) в Linux используются для синхронизации потоков и представляют собой альтернативу мьютексам, обеспечивая более быструю блокировку при условии короткого времени ожидания. Основное отличие спинлока от мьютекса заключается в том, что при блокировке ресурса спинлок не переводит поток в состояние ожидания, а продолжает активно проверять доступность ресурса в цикле, что снижает задержки, но увеличивает нагрузку на процессор.

Инициализация спинлока выполняется с помощью функции pthread_spin_init, где указывается, будет ли спинлок использоваться только потоками одного процесса (PTHREAD_PROCESS_PRIVATE) или потоками разных процессов через общую память (PTHREAD_PROCESS_SHARED). Разрушение спинлока производится через функцию pthread_spin_destroy.

Блокировка ресурса осуществляется функцией pthread_spin_lock, которая повторяет попытки до тех пор, пока ресурс не станет доступным. Для проверки

доступности ресурса без ожидания можно использовать pthread_spin_trylock, которая возвращает ошибку EBUSY, если ресурс заблокирован. Разблокировка ресурса выполняется функцией pthread_spin_unlock.

49. Механизмы синхронизации в операционной системе Linux. Блокировки читателей/писателей.

RW-lock (блокировка чтения-записи, pthread_rwlock_t) — это примитив синхронизации в Linux, предназначенный для решения проблемы «много читателей, один писатель». Этот механизм позволяет множеству потоков одновременно читать данные, но обеспечивает эксклюзивный доступ для записи, предотвращая изменения данных, пока они используются другими потоками. RW-lock является более производительным инструментом, чем мьютексы, поскольку он устраняет избыточные блокировки при операциях чтения

RW-lock инициализируется функцией pthread_rwlock_init(), принимающей указатель на объект блокировки и атрибуты. Если настройки не требуются, можно передать NULL в качестве атрибутов. Также возможно использование статического конструктора PTHREAD_RWLOCK_INITIALIZER для упрощённой инициализации.

Для потоков-читателей используются функции pthread_rwlock_rdlock() (блокировка) и pthread_rwlock_unlock() (разблокировка). Блокировка записи обеспечивается функциями pthread_rwlock_wrlock() и pthread_rwlock_unlock(). Существуют также неблокирующие варианты, такие как pthread_rwlock_tryrdlock() и pthread_rwlock_trywrlock(), которые немедленно возвращают код ошибки, если ресурс занят.

50. Виртуальная память в операционной системе Linux.

Виртуальная память в операционной системе Linux представляет собой метод управления памятью, который позволяет приложениям использовать больше памяти, чем физически доступно в системе. Эта технология обеспечивает изоляцию процессов, оптимизацию использования ресурсов и защиту данных. Основной принцип работы виртуальной памяти основан на трансляции виртуальных адресов, генерируемых приложениями, в физические адреса с помощью таблиц страниц и ассоциативной памяти (TLB).

Виртуальная память организована через страничную или сегментно страничную схему. При страничной организации память делится на страницы фиксированного размера (обычно 4 КБ). Каждая страница может быть выгружена на диск при недостатке оперативной памяти и подгружена обратно при необходимости, что позволяет экономно использовать физические ресурсы. Сегментно-страничная схема объединяет логическое разбиение на сегменты и страничное отображение, обеспечивая гибкость при распределении памяти.

Таблица страниц является ключевым компонентом управления виртуальной памятью. Она сопоставляет виртуальные страницы с физическими кадрами, а

её структура может быть многоуровневой для оптимизации хранения данных. Например, двухуровневая таблица страниц использует таблицы первого и второго уровней для уменьшения объёма данных, необходимых для адресации. Ассоциативная память (TLB) позволяет ускорить поиск адресов, временно храня часто используемые записи из таблицы страниц.

51. Автоматизация задач и управление системными сервисами в операционной системе Linux.
Автоматизация задач и управление системными сервисами в ОС Linux — это ключевой процесс, направленный на упрощение и улучшение работы с системами и сервисами. Он охватывает настройку инструментов и написание скриптов для выполнения рутинных операций, таких как резервное копирование, обновление пакетов и мониторинг состояния систем. Использование инструментов планирования, таких как cron, позволяет запускать задачи автоматически в определенное время, что исключает необходимость ручного выполнения. Автоматизация конфигурации с помощью Ansible, Chef или Puppet значительно упрощает развертывание и управление системами, особенно в микросервисной архитектуре.

Для управления системными сервисами в Linux широко используется systemd, который предоставляет централизованный интерфейс для запуска, остановки, перезапуска и мониторинга сервисов. Команды, такие как systemctl start, systemctl stop, и systemctl status, позволяют эффективно управлять состоянием служб. Благодаря journalctl администраторы могут анализировать логи, следить за работой системных сервисов и выявлять ошибки.

Задачи к экзамену «Операционные системы»

Все задания выполняются в консоли ОС Линукс и не требуют установки дополнительных пакетов и специальной настройки ОС.

1) Скопируйте все файлы из текущей директории в директорию backup, сохраняя их структуру и права доступа. Если директории backup не существует, создайте её.
mkdir backup - команда создания директории
Можно использовать -p флаг для сохранения прав доступа:
-r флаг для рекурсивного копирования директории
cp -rp /home/my_home /media/backup/my_home

2) Выведите список всех процессов текущего пользователя, отсортированный по использованию вычислительных ресурсов процессора.

Команда top -u user выведет список всех процессов текущего пользователя, после этого нажмите shift+p чтобы отсортировать по затратам процессора. Если вы забыли используйте команду man top.

top -u "$USER" -o %CPU

3) Создайте в текущей директории 10 пустых файлов с именами file1, file2, ..., file10 за одну команду.
for i in {1..10}; do touch file$i; done
или
touch file{1,2,3,4,5,6,7,8,9,10}

4) Переместите все файлы с расширением .log из текущей директории в директорию logs, создавая её при необходимости.
mkdir -p /home/admin/logs & cp *.log /home/admin/logs
создаём директорию & копируем файлы в неё

5) Удалите из файла data.txt все строки, содержащие слово error, сохранив результат в том же файле.
grep -v "error" data.txt > tmpfile && mv tmpfile data.txt
grep с флагом -v выбирает только строки без слова “error” после чего результат записывается во временный файл и уже его содержимое копируется в изначальный.

6) В файле numbers.txt содержатся числа, записанные по одному в строке. Выведите их сумму.
paste -sd+ numbers.txt | bc
или
awk '{n += $1}; END{print n}' numbers.txt

7) Подсчитайте общее количество строк во всех текстовых файлах (*.txt) текущей директории и её подкаталогов.
find . -name "*.txt" -exec wc -l {} \; | awk '{s+=$1} END {print s}'

find . -name "*.txt": Эта часть команды ищет все файлы с расширением .txt в текущей директории (.) и всех её подкаталогах.

-exec wc -l {} \;: Для каждого найденного файла выполняется команда wc -l, которая подсчитывает количество строк в файле. {} заменяется на имя найденного файла.

| awk '{s+=$1} END {print s}': Результат выполнения команды wc -l (количество строк для каждого файла) передается в awk, который суммирует эти значения и выводит общую сумму.

8) Проверьте, существует ли файл с именем config.json в текущей директории. Если он существует, выведите сообщение "Файл найден", иначе — "Файл не найден".

1. Создаем исполняемый файл: touch script.sh
2. Делаем файл исполняемым: chmod +x script.sh
3. Заходим в файл: nano script.sh
4. В файле пишем

```
#!/bin/bash

if test -f “config/json”
        then
        echo “Файл найден”
        else
        echo “Файл не найден”
fi
exit 0;
```

5. Выходим из файла и запускаем его: bash script.sh

9) Выведите список всех пользователей, зарегистрированных в системе, используя содержимое файла /etc/passwd.

```
cut -d ':' -f 1 /etc/passwd
```

Объяснение команды:

- cut: Это команда для извлечения определенных частей из строк.
- -d ':': Указывает, что разделителем является двоеточие (:).
- -f 1: Указывает, что нужно извлечь первое поле (имя пользователя).
- /etc/passwd: Указывает файл, из которого нужно извлечь данные.

НЕ ПРОВЕРЯЛ (вроде работает)

10) Найдите все файлы в текущей директории и её подкаталогах, изменённые в течение последних 24 часов.

```
find . -type f -mtime -1
```

Пояснение команды:

- find: команда для поиска файлов и директорий.
- точка указывает, что поиск будет выполняться в текущей директории.
- -type f: указывает, что нужно искать только файлы (не директории).
- -mtime -1: ищет файлы, изменённые в течение последних 1 дня (24 часов). Знак - перед 1 указывает на "меньше чем".

11) Подсчитайте количество символов, строк и слов в файле README.md.

```
wc README.md
```

Пояснение
Команда wc выведет три числа и имя файла:

- Первое число — количество строк.
- Второе число — количество слов.
- Третье число — количество байтов (символов).

12) Узнайте, сколько оперативной памяти используется в данный момент. Выведите эту информацию в человекочитаемом формате.
free -h
Пояснение

- free: команда для отображения информации о памяти в системе.
- -h: выводит информацию в человекочитаемом формате (например, в мегабайтах или гигабайтах).

Второй столбик(used) кол-во используемой памяти

13) Создайте символическую ссылку на файл script.sh с именем run.sh.
Команда
ln -s script.sh run.sh

Пояснение

- ln: команда для создания ссылок.
- -s: флаг, указывающий на создание символической (или мягкой) ссылки.
- script.sh: имя исходного файла, на который вы хотите создать ссылку.
- run.sh: имя символьной ссылки, которую вы хотите создать.

Проверка
Чтобы убедиться, что символическая ссылка была создана, вы можете использовать команду ls -l:
ls -l run.sh

Вы должны увидеть что-то вроде этого:
lrwxrwxrwx 1 user user 10 date time run.sh -> script.sh

14) В файле contacts записаны контактные данные сотрудников на английском языке.
Выведите в терминал список только тех номеров телефонов, которые помечены тегом рабочий (work) или личный (private).
Формат записи данных в файле: <фамилия> <имя> <тег> <номер_телефона, начинающийся на +>
grep -E 'work|private' contacts.txt | awk '{print \$4}'

Объяснение команды:

- grep -E 'work|private' contacts.txt: Эта команда ищет строки, содержащие либо "work", либо "private" в файле contacts.txt.

- |: Этот символ передает вывод предыдущей команды в следующую команду.
- awk '{print \$4}': Эта команда извлекает и выводит четвертое поле (номер телефона) из строк, которые были отфильтрованы командой grep. Поля разделяются пробелами по умолчанию.

15) В файлах 1,2,3,4,5,6,7,8,9 каталога tests содержатся результаты экспериментов.
К сожалению, в ходе записи результаты были перемешаны и распределены по файлам в случайном порядке, часть результатов была продублирована. Отсортируйте результаты в порядке возрастания и запишите в файл results в каталоге reports.
(каталоги tests и reports лежат в текущей директории)
Записи в файлах хранятся в формате: # <номер эксперимента> : <результат эксперимента>
(слова с лекции: вытащить файлы в один пул и отсортировать результаты будем записывать через перенаправления вывода, то есть записи в файл. всего 3 команды)
cat tests/1 tests/2 tests/3 tests/4 tests/5 tests/6 tests/7 tests/8 tests/9 | sort -u > reports/results

Объяснение команды:
- cat tests/1 tests/2 tests/3 tests/4 tests/5 tests/6 tests/7 tests/8 tests/9: Эта команда объединяет содержимое всех файлов с результатами экспериментов.
- |: Этот символ передает вывод команды cat в следующую команду.
- sort -u: Эта команда сортирует строки в порядке возрастания и удаляет дубликаты (-u означает уникальные строки).
- > reports/results: Эта часть перенаправляет итоговый вывод в файл results, который будет находиться в каталоге reports.

НЕ ПРОВЕРЯЛ

16) В файле forecast содержится прогноз погоды на текущий месяц в формате
"...
14.01:солнечно, 5°C
16.10:пасмурно, 12°C
..."
Выведите прогноз на текущий день.
(Для 14 января вывод: Сегодня солнечно, 5°C)

today=$(date +'%d.%m') # Получить текущую дату в формате DD.MM
grep "^$today:" forecast | sed "s/^$today://; s/^/Сегодня /"

Объяснение команд:

- today=$(date +'%d.%m'): Эта команда сохраняет текущую дату в переменной today в формате ДД.ММ.
- grep "^$today:" forecast: Ищет строку в файле forecast, которая начинается с текущей даты. Символ ^ указывает на начало строки.
- sed "s/^$today://; s/^/Сегодня /":
    - s/^$today:// — удаляет дату из начала строки.
    - s/^/Сегодня / — добавляет "Сегодня " в начало строки.

17) Удалите один из файлов текущей директории, содержащий строку delete me

grep -l "delete me" * | head -n 1 | xargs rm

Объяснение:

- grep -l "delete me" *: Ищет файлы в текущей директории, которые содержат строку "delete me". Флаг -l выводит только имена файлов.
- head -n 1: Берет только первый найденный файл из списка.
- xargs rm: Передает имя файла команде rm для его удаления.

18) Посчитайте и выведите в терминал количество каталогов, расположенных в директории на два уровня выше данной.

find ../../ -maxdepth 1 -type d | wc -l

Объяснение:

- find ../../: Эта команда ищет в директории на два уровня выше текущей. Два точки (..) обозначают уровень вверх, а два уровня — это ../../.
- -maxdepth 1: Ограничивает поиск только текущим уровнем в указанной директории, то есть не углубляется в подкаталоги.
- -type d: Указывает, что нужно искать только каталоги.
- | wc -l: Передает результат команды find в wc (word count), который подсчитывает строки. Поскольку каждая строка соответствует одному найденному каталогу, это дает общее количество каталогов.

19) Запишите вывод команды ls в файл list
(Файл list не должен содержать своего имени в списке, других файлов с таким именем в директории нет)

ls | grep -v '^list$' > list

Объяснение:

- ls: Эта команда выводит список файлов и каталогов в текущей директории.
- |: Этот символ используется для передачи вывода одной команды на вход другой.
- grep -v '^list$': Эта команда фильтрует вывод, исключая строки, которые точно совпадают с именем list. Здесь:
    - -v означает "инвертировать" (т.е. исключить совпадения).
    - '^list$' — это регулярное выражение, которое соответствует строке, содержащей только list (начало и конец строки).
- > list: Перенаправляет (записывает) отфильтрованный вывод в файл с именем list.

20) Выведите имя родительского каталога текущей директории (воспользуйтесь выводом команды pwd). Если команда pwd для данной директории выводит сообщение вида: /home/work/task_1
Искомая команда должна выводить сообщение вида: work
basename $(dirname $(pwd))
basename выделяет из переданного пути последнюю часть
dirname обрезает от переданного пути последнюю часть, остаавляя всё остальное

21) Создайте каталог animals с двумя подкаталогами mammals и birds. Создайте в каталоге mammals пустой файл bear, а в birds - penguin. Удалите все созданные файлы и каталоги. Уложите решение в 3 команды.
mkdir animals animals/mammals animals/birds
touch animals/mammals/bear animals/birds/penguin
rm -r animals
-r – рекурсивное удаление

22) Выведите список файлов текущей директории с расширением txt, в именах которых третий с конца символ названия – x или t
find -name '*t??.txt' -o -name '*x??.txt'
-o это логическое или
-name критерии поиска

23) Запишите сегодняшнюю дату в формате dd-mm-yy в переменную TODAY.
Выведите ее в терминал.
TODAY=$(date '+%d-%m-%y')
echo $TODAY

Создаётся переменная TODAY с типом date и форматом dd-mm-yy

echo выводит переменную

24) Напишите приветственную команду. Команда должна выводить плакатными буквами сообщение вида:
Hello, <имя_текущего_пользователя>!
figlet - библа
sudo apt install figlet – Устанавливаем fglet
figlet "Hello, "$(whoami)
whoami – возвращает имя текущего пользователя
(Хз то ли это, что Островский хочет, возможно вот этот figlet нужно сделать приветственной командой)

25) Создайте файл с именем *. Удалите только этот файл.
touch '*'
rm '*'
Должно сработать (у меня получилось)

Мб лучше будет так:
touch *
rm *

26) Измените права доступа к файлу data.log, чтобы только владелец мог читать и записывать его, а остальные пользователи не имели доступа.
chmod o-rwx data.log
(эта команда отнимает у всех пользователей кроме владельца права на запись, чтение и исполнения файла, не знаю удовлетворит ли это Островского)

27) Создайте директорию projects и измените её права доступа так, чтобы все пользователи могли создавать файлы в этой директории, но удалять можно было только свои файлы.
mkdir projects
chmod -R o-w projects
-R – рекурсивно изменяет настойки созданных файлов
o-w запрещает пользователям изменять не свои файлы

28) Измените владельца файла report.txt на пользователя user1 и группу владельцев на group1.
sudo useradd user1 – Создаём пользователя user1
sudo groupadd group1 – создаём группу пользователей group1
sudo chown user1:group1 report,txt – меняем владельца и группу владельцев

29) Создайте файл secure.txt и запретите его чтение, запись и выполнение всем, кроме владельца.
touch secure.txt – создаём файл
chmod o-rwx secure.txt – меняем права
o – все пользователи кроме владельца
rwx – чтение запись и исполнение файла

30) Найдите все файлы в текущей директории и её подкаталогах, у которых отсутствуют права на чтение для группы.
find . ! -perm /g+rwx
find ищет в . текущем директории и её подкаталогах файлы и директории ! без -perm прав доступа g для группы на rwx чтение запись и исполнение.

31) Создайте группу developers и добавьте в неё текущего пользователя.
sudo groupadd developers – создание группы developers
whoami – узнать какой сейчас пользователь
sudo usermod -aG developers <имя_пользователя> – добавление пользователя в группу
(если найдёте способ добавить пользователя за 1 команду, не узнавая его текущее имя – поменяйте)

32) Измените права доступа ко всем файлам с расширением .conf в текущей директории, сделав их доступными для чтения только для владельца и группы.
chmod o-r *.conf - забираем у остальных пользователей права на чтение
chmod u=r *.conf - пользователь получает права на чтение и только на чтение.
chmod g=r *.conf - группа получает права на чтение и только на чтение.

33) Создайте файл shared.txt и разрешите всем пользователям изменять его содержимое.
touch shared.txt – создаём файл
chmod a+w shared.txt – меняем права
a – для всех пользователей
+w – добавить права на запись
o - это для остальных для всех это a - all (так то создатель изначально имеет право на редактирование)

34) Проверьте, имеет ли текущий пользователь право на выполнение файла script.sh. Если права отсутствуют, выдайте соответствующее сообщение.
whoami – узнать текущего пользователя
stat -c %U script.sh – узнаём кто владелец файла и какие права доступа к файлу

если пользователь владелец файла и права на выполнение отсутствуют:

chmod u+x script.sh – даём владельцу(u) права на выполнение (+x) если пользователь не владелец файла то:

sudo chmod o+x script.sh – даём всем остальным(o) права на выполнение (+x)

или:

sudo chmod g+x script.sh – группе текущего пользователя(g) права на выполнение (+x)

35) Создайте директорию restricted и запретите доступ к её содержимому для всех, кроме владельца.

mkdir restricted - создаём директорию

chmod o= restricted - устанавливаем права для остальных без возможности чтения, записи или доступа.

36) Настройте файл readonly.txt так, чтобы он не мог быть удалён даже владельцем, но мог быть прочитан.

sudo chattr +i readonly.txt - chattr устанавливает новые атрибуты для файла. Атрибут i делает файл неизменяемым, его нельзя удалить, но можно прочитать.

37) Проверьте наличие установленного ограничения на количество открытых файлов для текущего пользователя. Выведите это значение в терминал.

ulimit -Hn – выводит hard-cap для кол-ва открытых фйлов

38) Создайте пользователя testuser с домашним каталогом /home/testuser и паролем password123.

sudo adduser testuser – создаём пользователя, после 2 раза вводим пароль: password123, не бойтесь, что при вводе пароля он не отображается, всё работает, после прокликиваете enter, вводите y и всё

39) Определите, какие файлы в директории /etc принадлежат группе root и имеют права на выполнение.

find /etc -group root -perm /a+x – find поиск в /etc директории файлов к которым -group group имеет право на -perm /a+x выполнение.

40) Удалите файл test.log, если он существует, только если текущий пользователь является его владельцем.

find . -type f -name "test.log" -user <имя_пользователя> -exec rm -f {} \; - находим файл по критериям среди которых проверка на владельце и удаляем через -exec rm -f {} \;

41) Измените права доступа к файлу script.sh, чтобы он мог выполняться только в определённое время (например, с 9:00 до 17:00).
sudo apt install cron – устанавливаем chron
(crontab -l ; echo "0 9 * * * chmod a+x /script.sh; 0 17 * * * chmod a-x /script.sh") | crontab -

42) Узнайте текущее значение переменной среды PATH. Добавьте к ней путь /opt/bin и выведите обновлённое значение.
echo $PATH – Выводим значение PATH
export PATH=$PATH:/opt/bin – Добавляем к PATH /opt/bin
echo $PATH – Выводим значение PATH

43) Найдите все открытые файлы текущим пользователем, принадлежащие каталогу /var.
whoami – узнать текущего пользователя
sudo lsof +D /var -u <имя_пользователя>
lsof – выводи открытые файлы
+D – указывает на директорию
-u – указывает на пользователя
(Возможно это не правильно но это всё, что у нас есть)

44) Выведите дерево процессов текущей системы с указанием имени и идентификатора процесса.
ps -e --forest -o pid,comm

ps позволяет показать все процессы; -e: производится выборка всех текущих запущенных процессов; --forest выводит данные в виде дерева; после аргумента -o оставляем только поля, которые нужны: pid и команда запуска (имя процесса)

45) Установите лимит процессорного времени для текущей оболочки на 60 секунд и запустите программу stress для тестирования.
sudo apt install stress – устанавливаем stress
ulimit -t 60 && stress --cpu 4 --timeout 60

46) Найдите все файлы размером более 100 МБ в домашнем каталоге текущего пользователя.
find /home -type f -user <имя_пользователя> -size +100M - поиск с параметрами, разъяснения есть выше.

47) Удалите все файлы, которым исполнилось более 30 дней, из директории /tmp.

find /tmp -mtime +30 -type f -name * -exec rm -f {} \; - добавили параметр -mtime +30, который выбирает файлы которые модифицировались 30 и более дней назад. Остальное выше.

или

```
sudo find /tmp -type f -mtime +30 -delete
```

48) Создайте пользователя guest с ограниченным доступом к системе и удалите его через 7 дней.

useradd -e `date -d "7 days" +"%Y-%m-%d"` user_name- создаем пользователя с параметром -e, который определяет дату отключения. Дату задаем через "7 days" количество дней и "%Y-%m-%d" формат записи.

49) Просмотрите историю команд текущего пользователя, отфильтровав только команды, содержащие слово chmod.

history | grep chmod

history – команда, которая выводит историю терминала

grep – фильтр который используется для поиска по истории команд