Экзаменационные вопросы

### **1. ОС как базовая часть систем обработки данных.**

**Система обработки данных** - совокупность технических средств и ПО, предназначенная для информационного обслуживания пользователей и технических объектов.

Общая структура классификации СОД разделена на 2 больших класса:

* **сосредоточенные(централизованные) системы** - обработка данных ведется отдельной ЭВМ,вычислительным комплексом или вычислительной системой.
* **распределенные системы** - процессы обработки данных рассредоточены по многим компонентам: системам телеобработки или вычислительным сетям.

Каждый элемент такой классификации имеет свою ОС.

**ОС - это ПО**, которое устанавливается на аппаратную часть вычислительной техники.С этой точки зрения в отдельной ЭВМ различают:

* техническую часть - аппаратное обеспечение компьютера
* программную часть - программное обеспечение компьютера

**Архитектура компьютера** представляет набор функциональных компонент, во многом работающих независимо друг от друга, но согласованно взаимодействующих через общую системную магистраль.

**Системный блок** - основа архитектуры аппаратной части ЭВМ, в котором размещены:

1. микропроцессор
2. блок оперативного запоминающего устройства (озу)
3. микросхемы постоянного запоминающего устройства(пзу)
4. устройства долговременной памяти на жестком диске (Винчестер)
5. устройства для запуска компакт-дисков и дискет
6. интерфейсные платы: сетевая, видеопамяти, обработки звука, модем (модулятор-демодулятор), платы, обслуживающие устройства ввода-вывода: клавиатуры, дисплея, "мыши", принтера и другие устройства.

Программная часть ЭВМ разделяется на 3 категории:

* **системное ПО** - ОС и программы общего пользования,выполняющие различные вспомогательные функции, например, создание копий используемой информации, выдачу справочной информации о компьютере, проверку работоспособности устройств компьютера и другие
* **прикладное ПО** - программы, обеспечивающие выполнение необходимых работ на ЭВМ: редактирование текстовых документов, создание рисунков или картинок, обработка информационных массивов и другие
* **инструментальное ПО** - программы, обеспечивающие разработку новых программ для компьютера на различных языках программирования

**Системное ПО** - это программы общего пользования, не связанные с конкретным применением ЭВМ и выполняющие традиционные функции: планирование и управление задачами, управление вводом-выводом и другие. К нему относятся

1. ОС - программа, которая загружается в ОЗУ ЭВМ при включении компьютера
2. программы-оболочки - обеспечивают более удобный и наглядный способ общения с компьютером, по сравнению с командной строкой DOS
3. операционные оболочки - интерфейсные системы, которые используются для создания графических интерфейсов, мультипрограммирования и другие
4. драйверы - программы, предназначенные для управления портами периферийных устройств
5. утилиты - вспомогательные или служебные программы, которые предоставляют пользователю ряд дополнительных услуг

### **2. Серверные ОС и рабочие станции.**

контекст аппаратного обеспечения:

**сервер** - специализированный компьютер или оборудование для выполнения на нем сервисного ПО; ЭВМ с повышенной надежностью исполнения, имеющее сетевые устройства и предназначенное для непрерывной работы в течении длительного без выключения или перезагрузки ОС;

**рабочая станция** - ЭВМ, оборудование которой расширено устройствами мультимедиа и другими системами, предназначенная для решения определенного круга задач; наличие оборудования для работы в сети является необязательным требованием, но требования к возможности интерактивного взаимодействия с пользователем являются определяющими

контекст ПО или комбинация с контекстом аппаратного обеспечения ЭВМ, подразумевающие использование парадигмы «клиент-сервер»:

**сервер** - любая запущенная программа, ориентированная в прикладном плане на обслуживание запросов других программ — клиентов;

**рабочая станция** - ЭВМ, предназначенная для интерактивной работы с пользователем, на которой установлено клиентское программное обеспечение.

После появления сетевых ОС:

**сервер** (server) — дистрибутив ОС или ЭВМ, с установленным системным и прикладным ПО, ориентированные на выполнение функций сервера;

**рабочая станция** (desktop) — дистрибутив ОС или ЭВМ, предназначенная для интерактивной работы с пользователем, на которой установлено соответствующее клиентское прикладное программное обеспечение.

### **3. ОС как виртуальная машина.**

### **4. Многослойная структура ОС.**

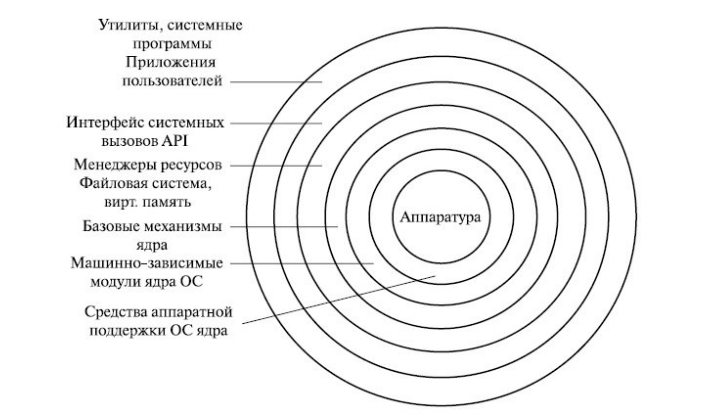
**Классическая архитектура ОС** основана на:

* концепции иерархической многоуровневой машины
* привилегированном ядре
* пользовательском режиме работы транзитных модулей.

**Модули ядра** выполняют базовые функции ОС:

* управление процессами, памятью
* устройствами ввода-вывода и тому подобное

В концепции многоуровневой иерархической машины структура ОС - ряд слоев. Внутренний слой обслуживает вышележащий слой через межслойный интерфейс.



Иерархическая архитектура ОС

Такая организация позволяет:

* сначала сверху вниз определить функции слоев и межслойные интерфейсы
* при детальной организации, двигаясь снизу вверх, можно наращивать мощность функций слоев
* модули каждого слоя можно изменять без необходимости изменений в других слоях, но не меняя межслойные интерфейсы

Повышение устойчивости работы ОС обеспечивается переходом ядра ОС в привилегированный режим

**Привилегированный режим** - особый режим работы процессора, поддерживаемый аппаратурой ЭВМ, в котором никакая программа, работающая в других режимах, не может прервать работу процессора.

### **5. ОС как базовая часть ПО ЭВМ.**

**Ядро ОС -** сердцевина системного ПО, без которого это ПО является полностью неработоспособным и не может выполнить ни одну из своих функций. В ядре решаются внутрисистемные задачи организации вычислительного процесса, недоступные для приложений. Таким образом, ядро ОС является базовым ПО ЭВМ.

Все приложения обращаются к ядру со специальными запросами - системными вызовами(пример: открытие и чтение файла;получение системного времени и т.д)

**API -** интерфейс прикладного программирования, который образуется функциями ядра.

### **6. Режимы ядра и пользователя.**

Ядро ОС работает в специальном привилегированном(защищенном) режиме для повышения надежности.

**Режим пользователя** - режим, в котором работают утилиты и остальное прикладное ПО ОС

### **7. Монолитное ядро и микроядерная архитектура ОС.**

**Монолитное ядро** ОС содержит ПО всех уровней, кроме последнего. Содержится в классической архитектуре ПО ОС UNIX.

Системный вызов такого ядра происходит в 2 этапа:

* системный вызов привилегированного ядра ОС инициирует переключение процессора из пользовательского режима в привилегированный
* при возврате к приложению происходит обратное переключение

За счет времени переключения 2t возникает дополнительная задержка в обработке системного вызова.



**Микроядерная архитектура** ОС - альтернатива классическому варианту архитектуры ОС. Суть состоит в следующем:

* В привилегированном режиме работает только очень небольшая часть ОС, называемая **микроядром**
* **Микроядро** защищено от остальных частей ОС и приложений
* в состав микроядра входят **машинно-зависимые модули** и **модули выполняющие базовые механизма обычного ядра**
* все остальные, более высокоуровневые функции ядра, оформляются как **модули, работающие в пользовательском режиме**

менеджеры ресурсов, вынесенные в пользовательский режим - сервера ОС.



Выполнение системного вызова сопровождается 4 переключениями режимов (4t), а значит при прочих равных условиях производительность ОС с микроядерной архитектурой ниже чем у ОС с классической.



### **8. Ядро и модули ОС.**

Модули ядра выполняют основные базовые функции ОС:

* управление процессами и памятью
* управление устройствами ввода-вывода и другими элементами

Хоть модульная организация ОС характерна для микроядерной архитектуры, монолитные ядра также используют модули, т.к аппаратная часть ЭВМ настолько многообразна, что практически неэффективно создавать ядро на все возможные варианты конфигурации модулей.

**драйверы** - специальное ПО, через которое осуществляется взаимодействие ядра ОС с аппаратной частью ЭВМ

Любое ядро ОС - одна большая программа,которая:

* выполняется в защищенном режиме(режиме ядра)
* выполняется в своем собственном адресном пространстве

Поэтому, перед компиляцией ядра запускается программа-конфигуратор, в которой можно указать какие драйвера включаются в ядро статически, а какие будут присутствовать как модули.

Для работы с модулями ОС Linux имеет специальные утилиты:

• lsmod — просмотр списка модулей;

• insmod — инсталляция модулей;

• rmmod — удаление модуля;

• modprobe — может выполнять функции insmod и rmmod;

• modinfo — получение информации о модуле.

### **9. Три базовых концепции ОС: файл, пользователь, процесс.**

**Концепция файла**

ОС UNIX (Linux) прямо декларируют парадигму: «Все есть файл».

К любому файлу потенциально применимы три операции: r - чтения, w - записи и x - запуска.

Конкретизация понятия файл, в аспекте хранилища данных, формализуется в понятие файловой системы, которая представляет собой поименованную совокупность обычных и специальных файлов.

**Обычный файл** — именованная упорядоченная последовательность байт. Специальный файл имеет имя и специализацию по назначению:

• устройства — отображение аппаратных средств компьютера в файловую систему ОС;

• директории — файлы, представляющие список имен файлов и директорий; обеспечивают иерархическую структуру файловой системы ОС. Операция запуск, применительно к директории означает возможность «войти в нее» пользователю;

• ссылки — именованные указатели на другие файлы, позволяющие работать с ними по имени ссылки.

• именованные каналы — точки доступа, через которые можно передавать данные;

• сокеты — точки доступа, через которые работает сетевое обеспечение ЭВМ.

**Концепция пользователя**

Все операции с файлами интерпретируются через концепцию пользователя.

Понятие пользователя интерпретируется через свои элементы - владелец, группа и другие:

• владелец — именованный и индексированный объект ОС, права которого интерпретируются на операции с файлами; индекс владельца или UID — User Identification — целое число (от нуля и выше), которое присутствует в каждом файле и интерпретируется как «хозяин файла»; чем меньше значение UID, тем более «важным» является пользователь; например, пользователь root имеет UID=0 и наивысшие права на файлы ОС;

• группа — именованный и индексированный объект ОС, предназначенный для объединения владельцев ОС; каждый владелец должен входить хотя бы в одну группу; индекс группы или GID — Group Identification, аналогичен UID и выполняет те же функции для групп; владелец root имеет собственную группу с именем root и идентификатом GID=0; все администраторы ОС обычно входят в группу root;

• другие — это пользователи без имени и идентификатора, которые дополняют концепцию пользователя и права которых также отображены в каждом файле.

**Концепция процесс**

Третья концепция - процесс, который обычно интерпретируется как запущенная программа или задача.

Процесс — это элементарный управляемый объект ОС, имеющий целочисленный идентификатор PID — Process Identification, обеспечивающий функциональное преобразование файлов (данных) с правами, которые определяются объектами пользователь.

Значения PID начинаются с 1 (обычно - это процесс init) — главный родительский процесс, и увеличиваются по мере порождения дочерних процессов.

Новому процессу присваивается номер на 1 больше, чем максимальный номер существующего или существовавшего с момента запуска ОС процесса

### **10. Системные вызовы fork(...) и exec(...).**

Запуск любой программы в режиме пользователя осуществляется с помощью двух системных вызовов: fork(...) и exec(...).



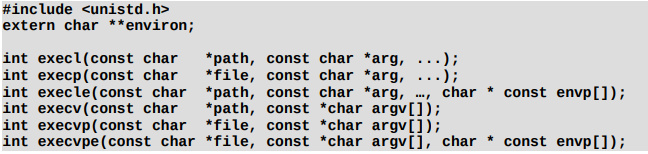
Функция fork(...) полностью дублирует существующий процесс, вместе со всеми открытыми файлами, порождая новый (дочерний) процесс с новым PID. Различают родительский и дочерний процессы только по целочисленному значению, которое возвращает функция fork(...):

-1 — ошибка, дочерний процесс не создан;

0 - дочерний процесс;

> 0 — родительский процесс, которому передано значение PID дочернего процесса.

Если дочерний процесс создан для запуска некоторой программы, то используется одна из разновидностей системной функции exec(...):



### **11. Дистрибутивы ОС.**

Когда говорят, что на ЭВМ установлена некоторая ОС, то обычно подразумевают некоторый ее дистрибутив, включающий конкретное ядро ОС и другое системное ПО, а также прикладное ПО и системы разработки.

Выбор конкретного дистрибутива предполагает учет многих факторов:

• производитель дистрибутива;

• тип процессора, на который рассчитан дистрибутив;

• лицензия дистрибутива и ценовая политика дистрибьютора;

• поддержка национальных языков;

• типы носителей, на которых распространяется дистрибутив;

• особенности инсталляции;

• сопровождение дистрибутива;

• наличие документации.

### **12. Архитектура х86.**

x86 (Intel 80x86) — архитектура процессора с одноименным набором команд, которая впервые была реализована в процессорах компании Intel

рагистры:

• для выполнения арифметических и логических операций служат регистры общего назначения: AX, BX, CX и DX;

• индексные регистры служат для формирования массивов; для этих же целей служат указательные регистры;

• регистр состояния содержит биты, которые изменяются в процессе выполнения различных операций;

• сегментные регистры являются указателями на начало областей оперативной памяти (сегменты);

• указатель команды — смещение относительно начала сегмента команд, определяемого регистром CS

### **13. BIOS и его функции.**

**BIOS** (Basic Input/Output System - базовая система ввода-вывода) - часть системного программного обеспечения ЭВМ, реализованная в виде микропрограмм, и обеспечивающая для ОС доступ к материнской плате компьютера.

В персональных IBM PC-совместимых компьютерах, использующих процессоры x86, BIOS записана в микросхему EEPROM (ПЗУ) и обеспечивает:

• начальное тестирование компьютера;

• последующую загрузку ОС

Пользователю предоставляется «Меню», которое позволяет выполнить некоторые специальные настройки компьютера. Для обеспечения выполнения настроек все указанные фирмы используют текстовый режим монитора, который именуется псевдографикой. Поскольку основные настройки BIOS выполняются самими производителями компьютера, то обычному конечному пользователю следует использовать только две возможности:

• установка приоритетов загрузочных устройств;

• установка адресов дополнительных плат расширения компьютера

Для сохранения настроек BIOS используется микросхема CMOS-памяти

В зависимости от версии BIOS и модели материнской платы, функции настройки BIOS могут меняться. В разных версиях, одни и те же функции могут иметь разные названия.

### **14. Этапы и режимы POST. UEFI и его стандартизация.**

Сразу после включения питания компьютера, во время начальной загрузки компьютера, при помощи программ записанных в BIOS, происходит самопроверка аппаратного обеспечения компьютера — POST.

**POST** (Power-On Self-Test) — самостоятельное тестирование после включения.

Может использоваться полный или сокращенный тест.

**Сокращенный тест**, включает четыре этапа:

1. Проверку целостности программ BIOS в ПЗУ, используя контрольную сумму.

2. Обнаружение и инициализацию основных контроллеров, системных шин и подключенных устройств: графического адаптера, контроллеров дисководов и другие. 3. Выполнение программ BIOS, обеспечивающих самостоятельную инициализацию внешних устройств.

4. Определение размера оперативной памяти и тестирования первого ее сегмента: 64 Кбайт.

**Полный регламент** работы POST:

1. Проверка регистров процессора;

2. Проверка контрольной суммы ПЗУ;

3. Проверка системного таймера и порта звуковой сигнализации;

4. Тест контроллера прямого доступа к памяти;

5. Тест регенератора генератора оперативной памяти;

6. Тест нижней области ОЗУ для проецирования резидентных программ в BIOS;

7. Загрузка резидентных программ;

8. Тест стандартного графического адаптера (VGA);

9. Тест оперативной памяти;

10. Тест основных устройств ввода (НЕ манипуляторов);

11. Тест CMOS - Complementary Metal-Oxide-Semiconductor;

12. Тест основных портов LPT/COM;

13. Тест накопителей на гибких магнитных дисках (НГМД);

14. Тест накопителей на жестких магнитных дисках (НЖМД);

15. Самодиагностика функциональных подсистем BIOS;

16. Передача управления загрузчику ОС.

### **15. Блочные и символьные устройства компьютера.**

Все файлы устройств разделяются на символьные и блочные:

* блочными называются внешние устройства компьютера, обмен данными с которыми можно производить только блоками: целостной упорядоченной последовательностью байт; к блочным устройствам относятся «жесткие» и floppy-диски, магнитные ленты, диски CDROM и другие; на блочных устройствах возможно создание файловых систем;
* все другие устройства, не являющиеся блочными, называются символьными; обмен данными с символьными устройствами осуществляется по одному байту; например, клавиатура, мышь, консоль экрана, COM-порты, сетевые устройства и другие — символьные.

Замечание

Магнитные ленты могут иметь физические блоки переменной длины. «Жесткие диски (вичестера)» имеют физические блоки фиксированной длины. Текущий стандарт физического блока винчестера: 1 сектор — 512 байт. Символьное устройство не обозначает текстового содержимого. Для символьных устройств, во многих случаях, понятие объема хранения данных не применимо.

ОС MS Windows обозначает разделы блочных устройств, имеющих форматы FAT и NTFS буквами с двоеточием: A:, B:, C:, …, Z:. Прописные и заглавные буквы — неразличимы. Символьные устройства обычно скрыты за графическим интерфейсом и, в явном виде, не используются.

ОС UNIX и Linux имеют общие правила обозначения устройств:

* имена устройств находятся в специальной директории /dev; в нее смонтирована специальная область ядра dev с файловой системой типа devtmpfs;
* имена устройств имеют имя драйвера, которое управляет этим устройством; прописные и заглавные буквы различаются;
* имена устройств, объединенных одним драйвером, разделяются цифрой, добавляемой к имени драйвера, начиная с нуля.

Замечание

При наличии соответствующих драйверов, в ОС UNIX и Linux, можно с блочным устройством работать как с символьным, поэтому понятия блочный и символьный применимы и к драйверам, управляющим устройствами.

### **16. Винчестер и загрузочные устройства.**

Традиционно, загрузочным устройством ЭВМ является винчестер или «жесткий диск».

Конструктивно, винчестер состоит из набора круглых пластин, которые центральной частью, на некотором расстоянии, надеты на шпиндель, вращающийся посредством электродвигателя:

* каждая сторона круглой пластины покрыта магнитным составом, способным фиксировать информацию посредством магнитных головок, которые «плавают» над каждой стороной диска;
* отдельная окружность на отдельной стороне диска образует трек (track);
* совокупность треков одного диаметра, образуют цилиндр (cylinder); цилиндры пронумерованы от внешнего края диска, начиная с 0;
* все магнитные головки винчестера (head) одновременно находятся над треками одного цилиндра и пронумерованы от 0 (обычно от 0 до 15);
* каждый трек разделен на 63 части — сегменты (физический блок), пронумерованные, начиная с 1.

Таким образом, физические блоки винчестера (сегменты) пронумерованы в системе координат CHS, начиная с сегмента (0, 0, 1).

Замечание

Следует отметить, что адресация CHS (Цилиндр, Головка, Сектор), заложенная в конструкцию первых персональных компьютеров и ПО BIOS, не позволяет адресовать более 7.8 Гбайт данных. Поэтому современные ЭВМ, имеющие винчестера емкостью более 7.8 Гбайт, используют адресацию LBA.

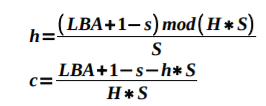
LBA (Logical block addressing) — механизм адресации и доступа к блоку данных на на «жестком диске», при котором системному контроллеру нет необходимости учитывать геометрию самого жесткого диска: количество цилиндров, сторон и секторов на цилиндре. Контроллеры современных IDE-дисков в качестве основного режима трансляции адреса используют LBA.

Суть LBA состоит в том, что каждый блок, адресуемый на жестком диске, имеет свой номер - целое число, начиная с нуля и далее:

LBA 0 = Цилиндр 0/Головка 0/Сектор 1

Преимущество метода адресации LBA — ограничение размера диска обусловлено лишь разрядностью LBA. В настоящее время, для задания номера блока используется 48 бит, что дает возможность адресовать (248) 281 474 976 710 656 блоков. Технический комитет X3T10 установил правила получения адреса блока в режиме LBA, при условии, что размер блока равен размеру сектора:





где:

с — номер текущего цилиндра;

h — номер текущей головки;

s — номер текущего сектора;

H — число головок;

S — число секторов на дорожке;

mod — операция взятия остатка от деления.

После того, как BIOS закончит начальный тест POST, она начнет просматривать блочные устройства ЭВМ с целью поиска загрузочного устройства ОС:

* Блочные устройства просматриваются в том порядке, который указан в настройках BIOS. Чтобы определить является ли устройство загрузочным, BIOS читает первый сектор блочного устройства и помещает его в ОЗУ ЭВМ. В компьютерах архитектуры IBM PC, этот адрес обычно 0000:7c00.
* Если сектор соответствует MBR — Master Boot Record, то BOIS передает управление его загрузочному коду: обычно командой long jump.
* Если структура прочитанного сектора не соответствует MBR, то проверяется следующее устройство.
* Если все просмотренные устройства не являются загрузочными, то BIOS: или перезапускает ЭВМ или загружает встроенный в BIOS интерпретатор языка BASIC (если он, конечно, — есть).

Замечание

BIOS рассматриватет flashUSB как загрузочное устройство, если:

* его ПО поддерживает такие устройства;
* имеется раздел MBR, который отмечен как загрузочный.

UEFI рассматривает flashUSB как загрузочное устройство, если оно имеет раздел, форматированный как FAT12/16/32, и в корне разделя имеется директория EFI.

### **17. Загрузочный сектор MBR, его назначение и архитектура.**

MBR (Master Boot Record) — это Главная загрузочная запись блочного устройства. Более точно — это код и данные расположенные в первом секторе блочного устройства, которые могут быть использованы для загрузки некоторой ОС.

В общем случае, под загрузчик MBR выделено 32 Кбайт винчестера или другого внешнего блочного накопителя. Если под загрузчик ОС используются все 32 Кбайт, то под MBR понимают весь этот загрузочный код. В этом случае, первые 512 байт называют MBS — Master Boot Sector или главным загрузочным сектором. Для операционных систем MS Windows, понятия MBR и MBS совпадают, так как вся MBR содержится в MBS и они рассматриваются как синонимы.

Замечание

MBR может не содержать загрузочного кода, если блочное устройство не является загрузочным. Более того, сам термин появился в те времена, когда:

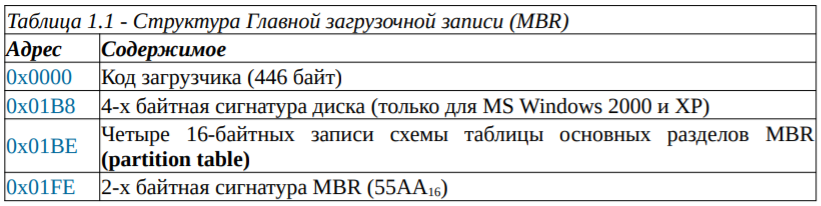
* с одного устройства загружалась только одна ОС;
* структура блочного устройства была уникальна для каждой ОС.

Последующая унификация структур блочных устройств и самих загрузочных записей привели к тому, что MBR — это еще не загрузка ОС, а всего лишь выбор: «с какого раздела жесткого диска следует загружать ОС»:

* На стадии MBR происходит только выбор раздела диска и ничего более.
* Загрузка самой ОС происходит на более поздних этапах.

Структура MBR содержит три основные части (см. таблицу 1.1):

* небольшой фрагмент исполняемого кода, - 446 байт;
* таблицу разделов (partition table);
* специальную сигнатуру.



Поскольку утвержденного стандарта на структуру MBR не существует, то используется «стандарт де-факто», распространенный Microsoft, и которого придерживаются большинство дистрибьютеров ОС.

Согласно «традиции MBR», винчестер может быть разбит на четыре основных раздела. Допускается один из разделов использовать как расширенный раздел и делить его дополнительно. Традиционно также, MBR создается или редактируется в момент инсталляции ОС на внешний носитель.

Когда BIOS прочитает первый сектор блочного устройства и запишет его по адресу 0000:7С00, она проверяет наличие сигнатуры 55AAh:

* Если сигнатура есть, управление передается коду загрузчика MBR;
* Если сигнатуры нет, то проверяется следующее блочное устройство.

Код загрузчика MBR:

* копирует себя с адреса 0000:7C00 по адресу 0000:6000, освобождая место для непосредственного загрузчика ОС;
* работает с таблицей разделов (partition table), структура отдельной строки которой показана в таблице 1.2: если загрузочный раздел найден, то первый сектор загрузчика ОС записывается по адресу 0000:7С00 и ему передается управление; если загрузочный раздел не найден или обнаружена ошибка записи partition table, то делается прерывание INT 18h и управление передается назад в BIOS.

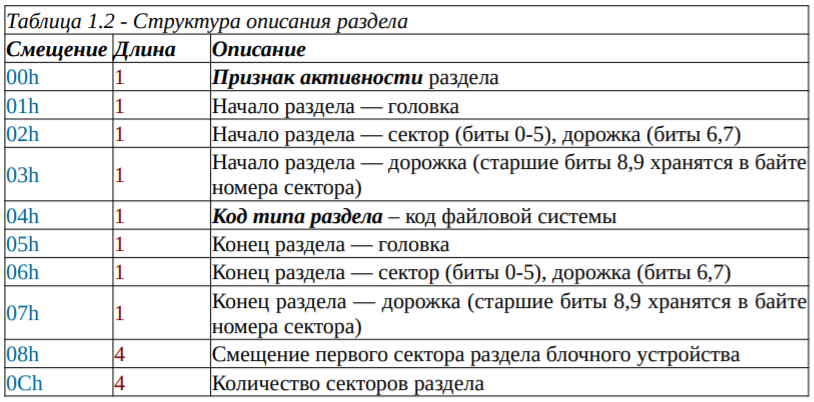
Для отдельной записи partition table:

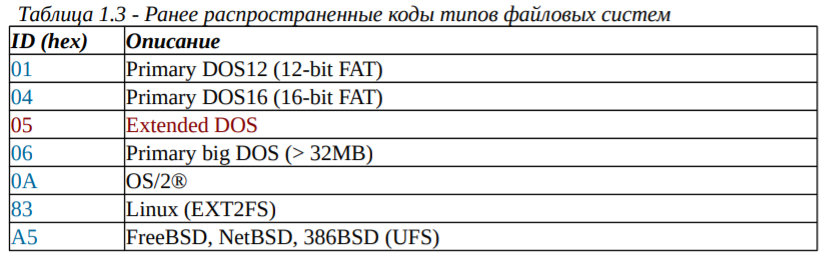
Первый байт содержит признак активности раздела: признак, обозначающий возможность загрузки операционной системы с данного раздела. Для стандартных загрузчиков может принимать следующие значения:

* 80h — раздел является активным;
* 00h — раздел является неактивным;
* Другие значения являются ошибочными и игнорируются.

Следующие три байта задают начало раздела в системе координат (C,H,S). Пятый байт обозначает код файловой системы: Partition Ids, некоторые значения которого приведены в таблице 1.3.

Следующие три байта задают окончание раздела в системе координат (C,H,S). Завершают строку partition table два четырехбайтовых числа, задающие начало раздела и его длину в секторах.





Замечание

Допускается, чтобы один из разделов блочного устройства имел код типа файловой системы равный 05h, который соответствует структуре раздела EBR — Extended Boot Record, начинающегося со структуры, приведенной в таблице 1.4.

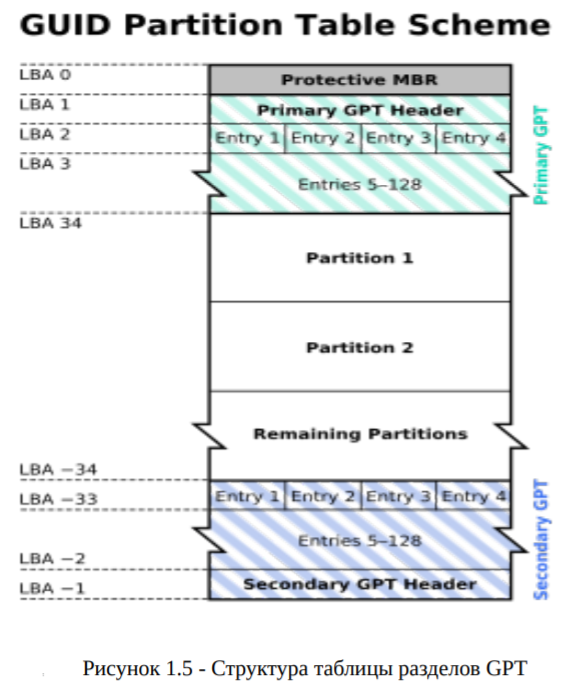


Замечание

Формат указателей в таблице 1.4 аналогичен формату строки Partition Table в MBR. Раздел EBR не может содержать в себе других разделов EBR. Традиционная таблица разделов винчестера MBR ориентирована на загрузку только одной ОС.

Дальнейшее развитие структуры блочных устройств связано с созданием новой усовершенствованной таблицы разделов: GPT, показанной на рисунке 1.6. GPT — GUID Partition Table. GUID — Global Unique Identifier - глобальный уникальный идентификатор, который в данном контексте используется для именования разделов блочных устройств:

* само устройство именуется в момент создания на нем струтуры GPT;
* раздел устройства именуется в момент создания на нем файловой системы.



Теперь,

* каждое блочное устройство может быть разбито на 128 разделов, записи о которых дублируются в конце устройства (отрицательные номера LBA).
* Все разделы GPT являются основными.
* Размер LBA=512 байт и LBA0 — для совместимости соответствует MBR.
* Отдельный LBA содержит записи о 4-х разделах: по 128 байт на раздел.
* Для идентификации раздела используются GUID.

Замечание

MBR, входящий в структуру GPT, должен указывать наличие на блочном устройстве только одного раздела с типом eeh и размером:

* всего блочного устройства, для устройств < 2 ТБайт;
* 2 ТБайт, для устройств > 2 ТБайт.

Не все BIOS могут работать со структурой GPT.

### **18. GRUB как универсальный загрузчик ОС.**

Проблема загрузки разных ОС с одного блочного устройства всегда будоражила умы пользователей ЭВМ, тем более, что емкость винчестеров постоянно увеличивалась. Хотя появились UEFI и структура блочных устройств GPT, имеется ряд проблем, в том числе и организационных, которые делают необходимым применение различных универсальных загрузчиков.

Наиболее распространенными загрузчиками ОС являются:

* NTLDR— загрузчик ядра MS Windows NT;
* Windows Boot Manager (bootmgr.exe, winload.exe) — загрузчик ядра MS Windows Vista; bootmgfw.efi — для MS Windows 7, 8, 8.1 и других;
* LILO (LInux LOader) — старый загрузчик ядра Linux;
* GRUB (Grand Unified Bootloader) — новый загрузчик ядра Linux и Hurd;
* RedBoot — загрузчик для встраиваемых систем;
* SILO (SPARC Improved bootLOader) — загрузчик Linux и Solaris для машин с архитектурой SPARC;
* Loadlin — загружает Linux из под MS DOS или MS Windows;
* Syslinux — загружает Linux из под MS DOS или MS Windows;
* BOOTP — применяется для загрузки по сети.

Среди перечисленного ПО, наиболее интересным является GRUB — официальный загрузчик Linux из проекта GNU. Он может загружать разные ОС, включая MS Windows, с многих разных аппаратных платформ. GRUB (точнее GRUB2) входит и в дистрибутив Arch Linux. Он устанавливается на ЭВМ в процессе инсталляции ОС. Это позволяет свободно использовать и Linux и MS Windows на одном компьютере.

Более подробно, вопросам работы с ПО GRUB посвящена лабораторная работа №2, которая входит в программу обучения по данной теме. В качестве дополнительного учебного материала, которым необходимо пользоваться как справочной информацией, является пособие [4, раздел 2].

### **19. Меню и функции GRUB**.

ПО GRUB можно рассматривать как маленькую однопользовательскую ОС специального назначения: интерактивная загрузка различных ОС.

Многие идеи этого ПО и, в частности модульная организация, использовались разработчиками UEFI.

Среди основных функций GRUB следует выделить:

* поддержка интерпретатора сценариев, близких по функциональным возможностям языка shell, который собственно говоря и был его прототипом;
* умение работать со структурами MBR и GPT блочных устройств;
* поддержка работы со многими современными устройствами ЭВМ;
* распознавание и умение работать с многими современными файловыми системами;
* поддержка национальных языков и других мультимедийных средств ЭВМ;
* обнаружение на блочных устройствах ЭВМ наличия различных ОС и автоматическое формирование для них сценария меню загрузки.

Хотя не все функции GRUB работают одинаково эффективно, потенциал этого ПО является очевидным.

Основная часть ПО GRUB располагается в разделе блочного устройства и устанвливается в процессе инсталляции на него ОС Linux. По умолчанию, оно помещается в директорию /boot/grub, туда же помещается автоматически созданный файл конфигурации grub.cfg.

Дополнительно, в процессе инсталляции ОС анализируется наличие UEFI и выполняется необходимая работа с ним.

Важно помнить, что дистрибутивы GRUB различаются для разных архитектур процессора ЭВМ, хотя допускается их смешанная установка на один компьютер.

Также, имеются варианты дистрибутивов для работы с ЭВМ, имеющими UEFI.

Замечание

При повторной инсталляции разных дистрибутивов ОС будет установлена и соответствующая версия ПО GRUB и частично могут быть изменены уже имющиеся его настройки. В первую очередь, это касается файла grub.cfg, поэтому следует заранее позаботиться о его сохранности.

Если на ЭВМ установлено несколько дистрибутивов ПО GRUB, то в процессе загрузки будет работать только один из них, хотя файл grub.cfg может быть общий!

Одним из вариантов использования ПО GRUB является установка его на flashUSB, что и применяется в процессе обучения по нашей дисциплине.

Такой подход позволяет избежать многих проблем, связанным с недостаточной квалификацией исполнителей.

### **20. Языки программирования и командные интерпретаторы**

Основным языком программирования ядра ОС и другого системного ПО является язык С. Этот язык специально создавался для написания ОС и постепенно вытеснил языки Ассемблера, которые, в большей степени чем С, зависели от архитектуры процессора, способов адресации памяти и других архитектурных особенностей ЭВМ.

С другой стороны, язык С также сильно привязан к архитектуре ЭВМ и ОС:

* через машинный язык, в который компилируется исходный текст языка С, исполняющийся конкретным процессором;
* через структуру исполняемых программ, которые определяются ОС;
* через библиотеку libc, связывающую, через системные вызовы, ПО режима пользователя с ПО ядра ОС.

Очевидно, что язык С мало пригоден для создания масштабных приложений. Для этих целей используются языки объектно-ориентированного программирования (ООП), такие как С++, C#, Java и другие.

Ранее отмечалось, что, являясь базовым ПО ЭВМ, ОС охватывает ту часть программного обеспечения компьютера, которое называется системным ПО.

Целевое назначение ОС — создание виртуальной машины или среды исполнения для работы системного, прикладного и инструментального ПО компьютера.

Важнейшая функция такой виртуальной машины — управление программным обеспечением ЭВМ, работающим в режиме пользователя.

Все ОС, для целей управления ПО ЭВМ, используют специальные языки программирования, которые называются командными интерпретаторами или shell:

* ОС MS Windows, в качестве shell, использует язык batch или cmd.
* ОС Linux — bash (Bourne Again Shell) и sh (Bourne Shell).
* ОС UNIX — sh (Bourne Shell), csh (C Shell), ksh (Korn Shell), tcl и другие.

Несмотря на имеющиеся различия, все командные языки имеют сходный синтаксис и построены по одному принципу - каждая строка языка рассматривается как команда с аргументами, требующая немедленного исполнения:

команда [ аргумент\_1 аргумент\_2 … ] конец\_строки

Строка — последовательность слов, разделенных символами пробела или табуляции и заканчивающаяся символами конца строки.

Команда — слово, обозначающее действие:

* встроенная команда исполняется непосредственно интерпретатором;
* имя программы ОС, которую интерпретатор запускает.

Аргумент — слово, интерпретируемое в контексте команды.

Конец\_строки — набор:

* управляющих символов языка
* управляющих слов языка.

Во времена, когда графический интерфейс ОС отсутствовал, командные интерпретаторы были единственным средством взаимодействия человека и ЭВМ. И сейчас они являются таковыми, когда графическая система выходит из строя.

Замечание

Работа любого командного языка опирается на специальное устройство ЭВМ, которое называется терминалом.

Терминал — это последовательное (символьное) устройство ЭВМ, обычно обозначаемое tty и обеспечивающее ввод с клавиатуры потока символов, обрабатывающее эти символы, а затем выводящее результат обработки на экран (дисплей) ЭВМ.

Именно терминал обеспечивает ввод команды с клавиатуры, отображение каждого символа на экране (дисплее) и передачу введенной строки интерпретатору shell, после нажатия клавиши «Enter».

По функциональным возможностям все языки приблизительно одинаковые, хотя в деталях могут различаться синтаксисом.

В частности, язык tcl разрабатывался с возможностью использования превдографики, что для своего времени было достаточно перспективно.

Совместное современное существование различных языков вызвано:

* силой привычки, авторскими правами и рядом корпоративных интересов;
* наличием достаточно большого количеста ПО, написанного на них.

Общая проблематика интерпретаторов заключается в том, что увеличение функциональных возможностей shell влечет:

* увеличение размера интерпретирующей прогаммы и уменьшение скорости ее загрузки;
* повышенный расход оперативной памяти компьютера.

### **21. Базовый язык shell (sh).**

Как отмечено выше, термин shell применяется в двух аспектах:

* как расширительное обозначение всех командных интерпретаторов ОС;
* как конкретизация интерпретатора sh (Bourne Shell).

Выбор языка sh обоснован следующими причинами:

* стандартизация языка в рамках проекта POSIX 1003.2 — стандарта мобильных систем;
* современные ядра ОС Linux запускают интерпретатор sh, при обнаружении в корне файловой системы скриптов (сценариев) init или linuxrc;
* интерпретатор bash, используемый ОС Linux, можно рассматривать как прямое функциональное расширение интерпретатора sh.

POSIX (Portable Operating System Interface for Unix) — переносимый интерфейс операционных систем UNIX.

POSIX — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой. Закреплен международным стандартом ISO/IEC 9945 и может использоваться не только для ОС UNIX.

Определим ряд метапонятий, которые shell учитывает в своей работе:

* shell — это программа (утилита или командный интерпретатор) sh, обычно /bin/sh, который работает в среде ОС: в пользовательском режиме;
* запустить sh может любой процесс, посредством одного из системных вызовов exec\*(); при этом, sh будет использовать среду ОС, в которой работала вызывающая программа;
* процесс sh может сам порождать необходимое количество дочерних процессов, посредством системного вызова fork(), отслеживая их работу и анализируя их коды завершения;
* нулевой целочисленный код завершения означает нормальное выполнение команды дочерним процессом;
* ненулевой целочисленный код завершения означает ошибочное выполнение команды дочерним процессом и дополнительно интерпретируется, в зависимости от ситуации и режимов работы sh.

Замечание

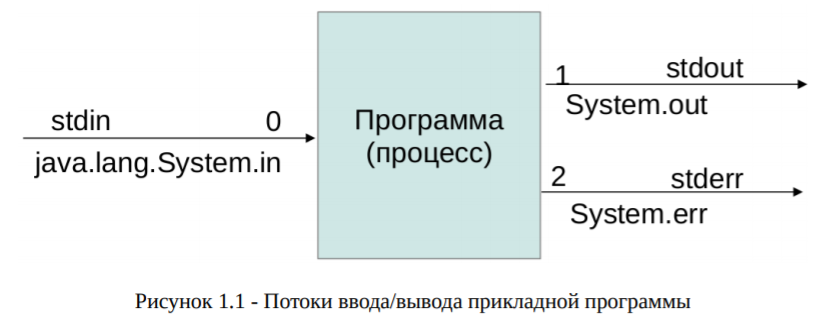
Если sh обнаружил синтаксическую ошибку, то выполнение shell прекращается, в противном случае, возвращается код завершения последней выполненной команды.

При запуске, sh как и любая прикладная программа, наследует все ресурсы вызывающего процесса, включая открытые файлы.

Замечание

Когда программа login завершит проверку входа пользователя в систему, она три раза открывает соответствующий терминал: один раз — на чтение и два раза — на запись, а затем запускает shell, обеспечивая системный ввод/вывод.

На рисунке 1.1 показан системный ввод/вывод типового прикладного процесса.



Каждая программа пользователя, запущенная в виде процесса на компьютере, имеет:

• один системный ввод;

• два системных вывода.

На уровне файловых дескрипторов shell, говорят об устройствах:

• устройство 0 — устройство ввода;

• устройство 1 — устройство нормального вывода программы;

• устройство 2 — устройство вывода ошибок.

На уровне языка С, мы имеем стандартные устройства:

• stdin — устройство ввода;

• stdout — устройство нормального вывода программы;

• stderr — устройство вывода ошибок.

На уровне языка Java, мы имеем три объекта:

• java.lang.System.in — объект канала ввода с клавиатуры;

• java.lang.System.out — объект канала нормального вывода;

• java.lang.System.err — объект канала вывода ошибок.

Для чтения из потока ввода (обычно клавиатура) используются различные модификации функции read(...).

Для вывода информации в потоки stdout и stderr (обычно консоль) используются различные модификации функций write(…) и print(...).

Замечание

Особо, следует обратить внимание на номера устройств (целочисленные дескрипторы файлов), которыми интенсивно манипулирует интерпретатор sh. Например, если процесс закрывает файл с дескриптором 0, а затем открывает новый файл. В результате, дескриптор нового файла будет равен 0 и процесс будет читать данные из файла, как будто он читает с клавиатуры.

Все shell используют свойства базовых категорий, определенных понятиями: файл, пользователь и процесс.

В частности, обычные файлы подразделяются на:

• бинарные, которые читаются процессом как последовательность байт, имеющих значения от 0 до 255;

• текстовые (символьные), в которых, в зависимости от кодировки, ряд значений байт или не используются совсем или рассматриваются как управляющие, например: 10 — перевод строки; 13 — возврат каретки.

Все интерпретаторы shell могут использовать текстовые файлы как программы. Такие файлы называют скрипты или сценарии.

В частности, все shell следуют общим правилам:

• символ # используется как комментарий до конца строки;

• сочетание символов #!, расположенных в первой позиции первой строки текстового файла, рассматривается как команда вызова конкретного интерпретатора shell.

Чтобы определить на каком терминале работает пользователь, необходимо воспользоваться командой ps, без аргументов.

Замечание

Операционная среда ОС Linux имеет 64 виртуальных терминала: /dev/tty0 … /dev/tty63. Переключение между терминалами /dev/tty1 … /dev/tty7 осуществляется комбинациями клавиш Alt-F1 … Alt-F7.

Программы (процессы), которые не имеют ввода/вывода на какой-либо терминал, называются системными процессами.

### **22. Среда выполнения программ.**

Среда выполнения любой программы ОС подразделяется на:

* структуру файловой системы ОС, которую программа использует для ввода и вывода данных;
* набор файлов конфигурации, которые определяют параметры данных программы или дополнительные данные конфигурации среды исполнения;
* системные переменные среды, которые наследуются как из среды родительского процесса, а также создаются или удаляются в процессе работы программы.

Для языка sh, среда исполнения определяется условиями видимости той части файловой системы, которая соответствует пользователю, запустившему shell. Как правило, для этих целей используется директория /home.

В течении работы и запуска интерпретатора sh используются следующие конфигурационные файлы: /etc/profile $HOME/.profile

Для интерпретатора bash, который является основным для обычных пользователей в ОС Linux, такими файлами являются: /etc/profile $HOME/.profile $HOME/.bash\_profile $HOME/.bash\_login $HOME/.bash\_logout $HOME/.bash\_history

Все интерпретаторы shell используют переменные среды, которые подразделяются:

* переменные системной среды — для системных процессов;
* переменные пользовательской среды — отдельно для каждого пользователя, вошедшего в систему

Sh использует следующие переменные пользовательской среды, где:

HOME Определяет домашний каталог пользователя. Подразумеваемый аргумент команды cd (1) - основной каталог.

PATH Список имен каталогов для поиска команд. Подобные списки называются списками поиска. Элементы списка разделяются: • двоеточием, для ОС UNIX; • точка с запятой для MS Windows; • точка - означает текущий каталог.

CDPATH Список поиска для команды cd.

MAIL Имя файла, куда будет помещаться почта; если переменная MAILPATH не определена, shell информирует пользователя о поступлении почты в указанный файл.

MAILCHECK Интервал между проверками поступления почты в файл, указанный переменными MAIL или MAILPATH. По умолчанию интервал составляет 600 секунд (10 минут). При установлении значения 0 проверка будет производиться перед каждым выводом приглашения.

MAILPATH Список имен файлов, разделенных двоеточием. Если переменная определена, shell информирует пользователя о поступлении почты в каждый из указанных файлов. После имени файла может быть указано (вслед за знаком %) сообщение, которое будет выводиться при изменении времени модификации указанного файла (сообщение по умолчанию "You have mail").

PS1 Основное приглашение (по умолчанию "$ ").

PS2 Вспомогательное приглашение (по умолчанию "> ").

IFS Цепочка символов, являющихся разделителями в командной строке (по умолчанию это пробел, табуляция и перевод строки).

SHACCT Если значением этой переменной является имя файла, доступного для записи пользователем, shell будет помещать в него сведения о каждой выполняемой им процедуре. Для анализа сведений могут быть применены такие программы, как acctcom (1) и acctcms (1M).

SHELL При запуске shell просматривает окружение в поисках этой переменной. Если она определена и файловая часть ее значения есть rsh, shell становится ограниченным [см. rsh(1)].

Замечание

Для переменных PATH, PS1, PS2, MAILCHECK и IFS имеются значения по умолчанию. Значения переменных HOME и MAIL устанавливаются командой login(1). Значения всех переменных можно вывести на консоль командой env.

Окружение [см. environ(3P)] - это набор пар (имя, значение), которые передается выполняемой программе так же, как и обычный список аргументов.

Shell взаимодействует с окружением несколькими способами:

* при запуске, окружение (среда работы) shell создается утилитой login, а затем интерпретатор передает это окружение всем запускаемым программам;
* присвоение значения какому-либо слову не оказывает никакого влияния на окружение, пока не будет использована команда export (см. также set -a);
* переменную среды можно удалить из окружения командой unset.

Таким образом, окружение каждой команды формируется из всех унаследованных языком shell:

* пар (имя, значение);
* минус пары, удаленные командой unset;
* плюс все модифицированные и измененные пары, к которым была применена команда export.

Окружение простой команды может быть модифицировано, если указать перед командой одно или несколько присваиваний переменным. Так, строки: TERM=vt100 команда и (export TERM; TERM=vt100; команда) являются эквивалентными, по крайней мере с точки зрения окружения команды.

### **23. Командная строка: опции и аргументы.**

Каждая строка символов, которая заканчивается символом «Enter», рассматривается терминалом как целостный набор данных, который необходимо передать интерпретатору shell. Сам интерпретатор анализирует полученную строку, проверяя ее синтаксис и разделяя ее на последовательность отдельных команд и их аргументов.

Первое слово в строке shell всегда воспринимается как команда, а остальные слова — как аргументы команды: • чтобы в явном виде разделить команды в строке, следует использовать разделитель — точка с запятой; • в случае, когда команда не помещается в одну строку, для продолжения ее на другой строке, используется символ — обратный слэш; • когда shell запускается посредством системного вызова exec\*(...) и первым символом нулевого аргумента является -, то сначала читаются и выполняются команды из файлов /etc/profile и $HOME/.profile.

Все команды shell, условно, разделяются на две группы: • встроенные команды — команды, которые интерпретатор выполняет самостоятельно; • внешние команды — это программы и утилиты, которые shell ищет в файловой системе и, после проверки прав доступа, пытается запустить, используя системные вызовы fork(...) и exec\*(...).

Кроме того, следует учесть что: • под пробелом, в дальнейшем, понимается не только собственно пробел, но также и символ табуляции; • имя - это последовательность букв, цифр и символов подчеркивания, начинающаяся с буквы или подчеркивания; • параметр - это имя, цифра или любой из символов \*, @, #, ?, -, $, !.

В общем случае, интерпретатор shell может рассматриваться как команда с аргументами. Синтаксис запуска интерпретатора shell имеет вид: sh [-a] [-c цепочка\_символов][-e][-f][-h][-i][-k][-n][-r][-s][-t][-u][-v] [-x] [аргумент ...] где - квадратные скобки обозначают необязательные конструкции.

Перечисленные флаги (опции) интерпретируются shell при его запуске следующим образом:

* если не указаны опции -s или -c, то первый аргумент рассматривается как имя файла, содержащего команды;
* остальные аргументы передаются этому командному файлу как позиционные параметры.

Наиболее часто используемые опции sh имеют следующую семантику:

-c цепочка\_символов Команды берутся из цепочки\_символов.

-s Если аргументов больше нет, то команды читаются со стандартного ввода. Все оставшиеся аргументы рассматриваются как позиционные параметры. Вывод сообщений самого shell, кроме специальных команд, направляется в файл с дескриптором 2 (стандартный протокол).

-i если ввод и вывод shell ассоциированы с терминалом, shell выполняется в интерактивном режиме. В этом случае сигнал завершения (0) игнорируется (то есть команда kill 0 не приведет к завершению работы интерактивного shell'а). Сигнал прерывания (2) перехватывается и игнорируется, поэтому выполнение системной функции wait (2) может быть прервано. В любом случае, сигнал выхода (3) игнорируется.

-r shell запускается как ограниченный [см. rsh(1)]

Описание остальных флагов и аргументов приведено в описании команды set.

### **24. Переменные shell.**

Все переменные sh, включая рассмотренные выше переменные среды, называются параметрами. Следует учесть,что:

• различаются два типа параметров: позиционные и ключевые;

• знак $ используется для подстановки значений параметра.

Позиционные параметры обозначаются цифрой или одним из символов: \*, @, #, ?, -, $, !.

Значения цифровых позиционных параметров устанавливаются при вызове shell функций или командой set:

• 0 — параметр 0 — имя вызываемой функции;

• 1 — параметр 1 — аргумент 1;

• 2 — параметр 2 — аргумент 2 и далее.

Значения следующих параметров shell устанавливает автоматически:

• \* или @ содержат все позиционные параметры, начиная с 1, разделенные пробелами;

• # количество позиционных параметров (десятичное);

• - флаги, указанные при запуске shell или установленные командой set;

• ? десятичное значение, возвращенное предыдущей командой;

• $ идентификатор процесса, в рамках которого выполняется shell;

• ! идентификатор последнего асинхронно запущенного процесса.

Ключевые параметры (переменные) обозначаются именами.

Значения им присваиваются обычным способом: имя=значение [имя=значение] …

Различаются следующие виды подстановок параметров:

${параметр}

Подставляется значение параметра, если оно определено. Скобки используются, только если за параметром следует буква, цифра или знак подчеркивания, и их нужно отделить от имени параметра. Вместо параметров \* и @ подставляются все позиционные параметры, начиная с $1, разделенные пробелами.

${параметр:-слово}

Будем говорить, что параметр пуст, если его значение не определено или является пустой цепочкой. При данном способе подстановки если параметр не пуст, подставляется его значение; в противном случае подставляется слово.

${параметр:=слово}

Если параметр пуст, ему присваивается слово; после этого подставляется значение параметра. Таким способом нельзя изменять значения позиционных параметров.

${параметр:?слово}

Если параметр не пуст, подставляется его значение; в противном случае в стандартный протокол выдается сообщение "параметр:слово" и выполнение shell'а завершается. Если слово опущено, то выдается сообщение "параметр:parameter null or not set".

${параметр:+слово}

Если параметр не пуст, то подставляется слово; в противном случае - не подставляется ничего.

Замечание

После проведения подстановок, полученная строка просматривается в поисках разделителей, которые берутся из системной переменной IFS, и расщепляется на аргументы.

Явные пустые аргументы сохраняются.

Неявные пустые аргументы удаляются.

Поскольку любой Shell интерпретирует команды и аргументы команд как слова, то следующие символы, если они не экранированы, завершают предыдущее слово:

; & ( ) | ^ < > пробел табуляция перевод\_строки

Символ \ используется для экранирования одиночных символов и удаляется из слова, перед выполнением команды, но сам экранируется одинарными кавычками.

Все эти символы могут экранироваться одинарными или двойными кавычками.

* двойные кавычки могут экранировать одинарную кавычку;
* двойные кавычки не мешают подстановке параметров.

### **25. Специальные символы и имена файлов.**

В командах, работающих с именами файлов, возможно использование шаблонов.

Шаблон — набор символов, который добавляет или изменяет имена файлов, используемые в командах интерпретаторов shell, как аргументы.

Типичные шаблоны, применяемые к именам файлов:

* \* сопоставляется с произвольной цепочкой символов, в том числе и пустой;
* ? сопоставляется с произвольным символом;
* [...] сопоставляется с любым, перечисленным в скобках символом. Пара символов, разделенных знаком -, рассматривается как отрезок алфавита. Если за символом [ стоит знак !, то шаблону удовлетворяет любой символ, не перечисленный в скобках.

Примеры использования шаблонов:

* ls .. - вывод списка файлов родительского каталога;
* ls . - вывод списка файлов текущего каталога (каталог, в котором находится пользователь);
* ls .\* - вывод всех списка файлов и списка содержимого каталогов, с именами начинающимися с «точки», для текущего каталога (каталог, в котором находится пользователь);
* ls .x\* - вывод списка имен файлов, начинающихся с .x, для текущего каталога;
* ls .[a-c,x]\* - вывод списка имен файлов, начинающихся с .a, .b, .c, .x, для текущего каталога;
* ls .config — вывод списка имен каталога .config;
* ls .config/\* — вывод списка имен файлов каталога .config и его каталогов.

### **26. Стандартный ввод/вывод и переадресация.**

В случае, когда для чтения и записи данных используются другие источники информации, применяются следующие правила перенаправления (переадресации) ввода и вывода:

<слово Использовать файл слово для стандартного ввода (дескриптор файла 0).

>слово Использовать файл слово для стандартного вывода (дескриптор файла 1). Если файла нет, он создается; если есть, он опустошается.

>>слово Использовать файл слово для стандартного вывода. Если файл существует, то выводимая информация добавляется в конец, то есть, сначала производится поиск конца файла; в противном случае файл создается.

<<[-]слово Читается информация со стандартного ввода, пока не встретится строка, совпадающая со словом, или конец файла. Если после << стоит -, то сначала из слова, а затем, по мере чтения, из исходных строк удаляются начальные символы табуляции, после чего проверяется совпадение строки со словом. Если какой-либо из символов слова экранирован, никакой другой обработки исходной информации не производится; в противном случае делается еще следующее: 1. Выполняется подстановка параметров и команд. 2. Пара символов \перевод\_строки игнорируется. 3. Для экранирования символов \, $, ` нужно использовать \. Результат описанных выше действий становится стандартным вводом команды

<&цифра Производить стандартный ввод из файла, ассоциированного с дескриптором цифра.

>&цифра Производить стандартный вывод в файл, ассоциированный с дескриптором цифра.

<&- Стандартный ввод команды закрыт.

>&- Стандартный вывод команды закрыт.

цифра<&- Закрыть ввод дескриптора цифра.

цифра>&- Закрыть вывод дескриптора цифра.

Если любой из этих конструкций предшествует цифра, она определяет дескриптор (вместо подразумеваемых дескрипторов 0 или 1), который будет ассоциирован с файлом, указанным в конструкции. Например, строка: ... 2>&1 ассоциирует дескриптор 2 (стандартный протокол) с файлом, связанным в данный момент с дескриптором 1. Важен порядок переназначения: shell производит переназначение слева направо. Так, строка: ... 1>f 2>&1 сначала ассоциирует дескриптор 1 с файлом f, а затем дескриптор 2 с тем же файлом. Если изменить порядок переназначения, стандартный протокол будет назначен на терминал (если туда был назначен стандартный вывод), а затем стандартный вывод будет переназначен в файл f.

Если команда состоит из нескольких простых команд, переназначение для всей команды будет выполнено перед переназначениями для простых команд. Таким образом, shell выполняет переназначения сначала для всего списка, за тем: для каждого входящего в него конвейера, для каждой команды конвейера, для каждого списка из каждой команды. Если команда заканчивается знаком &, то стандартный ввод команды переназначается на пустой файл /dev/null. В противном случае, окружение для выполнения команды содержит дескрипторы файлов запустившего ее shell'а, модифицированные спецификациями ввода/вывода.

### **27. Программные каналы.**

Рассмотренные выше метаопределения и элементарные понятия языка shell, опираются на понятие простой команды. Простая команда - это последовательность слов, разделенных пробелами:

• Первое слово определяет имя команды, которая будет выполняться, а оставшиеся слова передаются команде в качестве аргументов.

• Имя команды передается как аргумент 0 [см. exec(2)].

• Значение простой команды - это ее код завершения: 0 - если она выполнилась нормально, или (128 + код ошибки), если ненормально [см. также signal(2)].

Общие конструкции языка shell используют специальные файлы ОС, которые называются каналами (неименованными каналами).

Каналы — специальные файлы ОС, создаваемые посредством системного вызова pipe(...) и служащие для организации обмена данными (сообщениями) между процессами (программами).

В языке sh, для организации программных каналов между простыми командами, используется понятие конвейер.

Конвейер - это последовательность команд, разделенных знаком | (вертикальная черта). При этом:

* Стандартный вывод всех команд, кроме последней, направляется посредством системного вызова pipe(2) на стандартный ввод следующей команды конвейера.
* Каждая команда выполняется как самостоятельный процесс.
* shell ожидает завершения последней команды. Ее код завершения становится кодом завершения конвейера.

Список - это последовательность одного или нескольких конвейеров, разделенных символами ;, &, && или || и, быть может, заканчивающаяся символом ; или &.

Из четырех указанных операций:

* ; и & имеют равные приоритеты, меньшие, чем у && и ||.
* Приоритеты последних также равны между собой.
* Символ ; означает, что конвейеры будут выполняться последовательно.
* Символ & означает, что конвейеры будут выполняться параллельно (то есть shell не ожидает завершения конвейера).
* Операция && означает, что список, следующий за ней, будет выполняться лишь в том случае, если код завершения предыдущего конвейера нулевой.
* Операция || означает, что список, следующий за ней, будет выполняться лишь в том случае, если код завершения предыдущего конвейера ненулевой.
* В списке, в качестве разделителя конвейеров, вместо символа ; можно использовать символ перевод строки.

Двойные кавычки разрешают подстановки ключевых и позиционных параметров. Дополнительно, shell использует обратные кавычки: shell читает цепочки символов, заключенные в обратные кавычки, и интерпретирует их как команды. Такие команды выполняются в месте их использования. Например, abc=`ls -l`; здесь — ключевому параметру abc будет присвоен результат выполнения команды ls -l — список имен файлов текущей директории.

### **28. Сценарии.**

Простую команду или простой конвейер можно набрать и выполнить в окне консоли (терминала). Сложные конструкции языка sh — программы — пишутся в файлах, которые называются сценариями. Сценарий — последовательность простых команд и конвейеров, оформленных с помощью управляющих конструкций.

Язык sh содержит следующие управляющие конструкции (см. man sh). for имя [in слово ...] do список done При каждой итерации переменная имя принимает следующее значение из набора in слово .... Если конструкция in слово ... опущена, то список выполняется для каждого позиционного параметра.

case слово in [шаблон [| шаблон] ...) список ;;] ... esac Выполняется список, соответствующий первому шаблону, успешно сопоставленному со словом. Формат шаблона тот же, что и используемый для генерации имен файлов, за исключением того, что в шаблоне не обязательно явно указывать символ /, начальную точку и их комбинацию: элемент шаблона \* может успешно сопоставляться и с ними.

if список\_1 then список\_2 [elif список\_3 then список\_4] ... [else список\_5] fi Выполняется список\_1 и если код его завершения 0, то выполняется список\_2, иначе - список\_3 и если код его завершения 0, то выполняется список\_4 и т.д. Если же коды завершения всех списков, использованных в качестве условий, оказались ненулевыми, выполняется else-часть (список\_5). Если else-часть отсутствует, и ни одна then-часть не выполнялась, возвращается нулевой код завершения.

while список\_1 do список\_2 done Пока код завершения последней команды списка\_1 есть 0, выполняются команды списка\_2. При замене служебного слова while на until, условие продолжения цикла меняется на противоположное. Если команды из списка\_2, не выполнялись вообще, код завершения станавливается равным нулю.

(список) Группировка команд для выполнения их порожденным shell'ом.

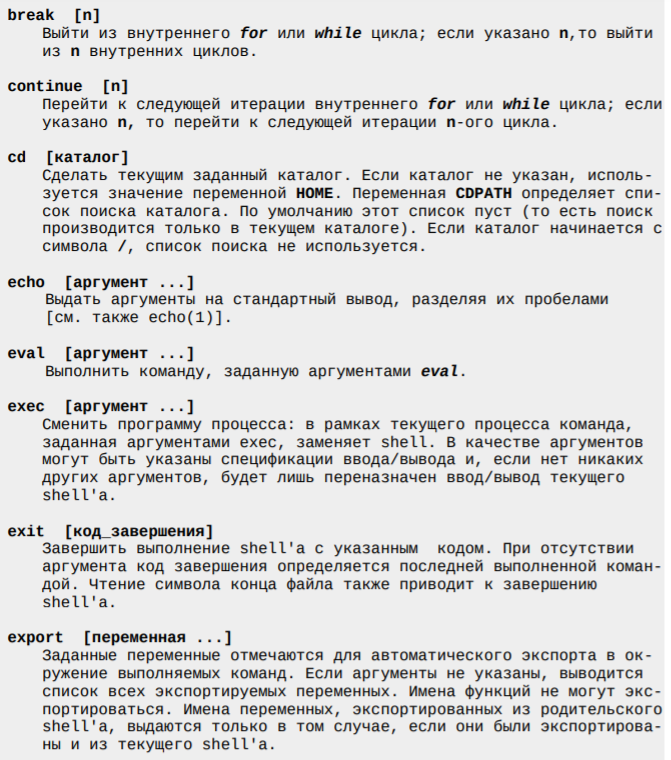
{список;} Группировка команд для выполнения их текущим shell'ом.

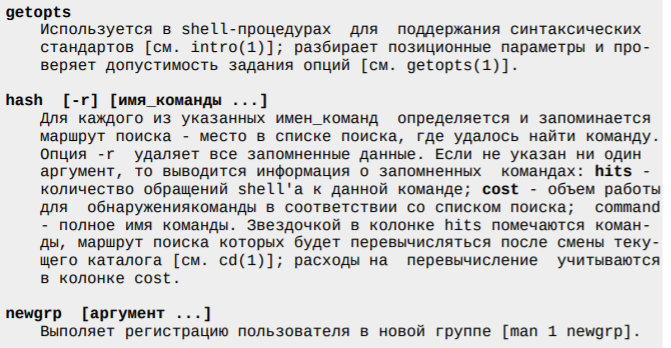
имя ( ) {список;} Определение функции с заданным именем. Тело функции - список, заключенный между { и }.

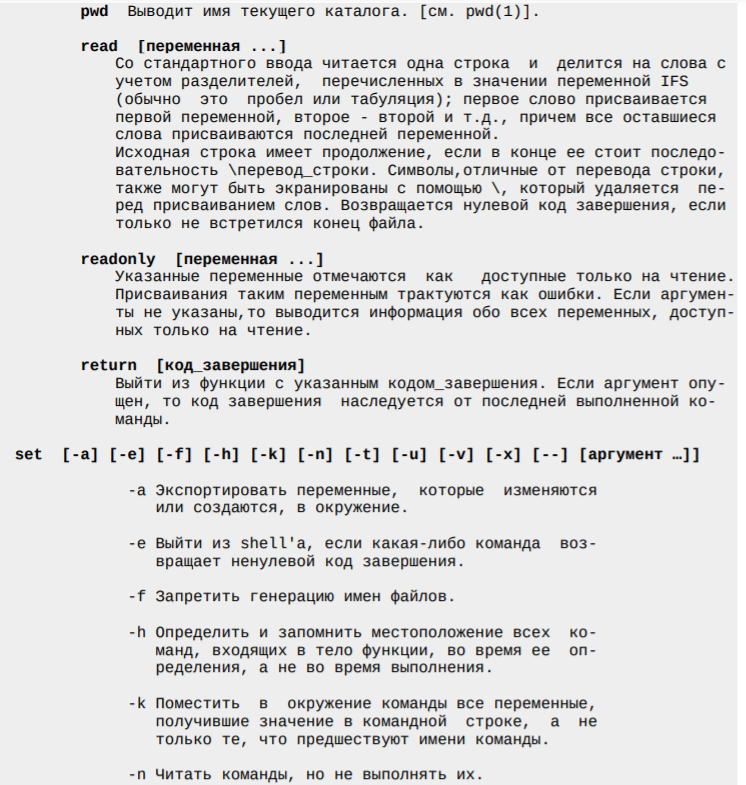
Следующие слова трактуются языком sh как ключевые, если они являются первым словом команды и не экранированы: if then elif else fi case in esac for while until do done { }

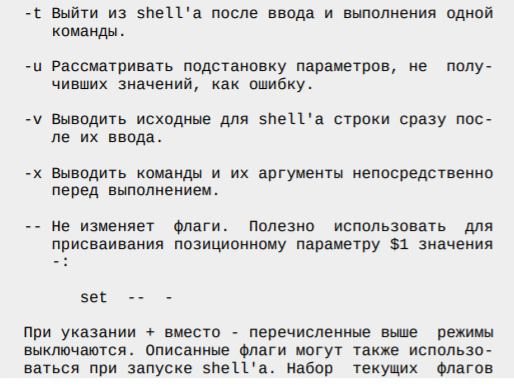
Кроме управляющих конструкций, sh содержит ряд встроенных (специальных) команд. Если не оговорено иное, команды выводят результаты в файл с дескриптором 1.

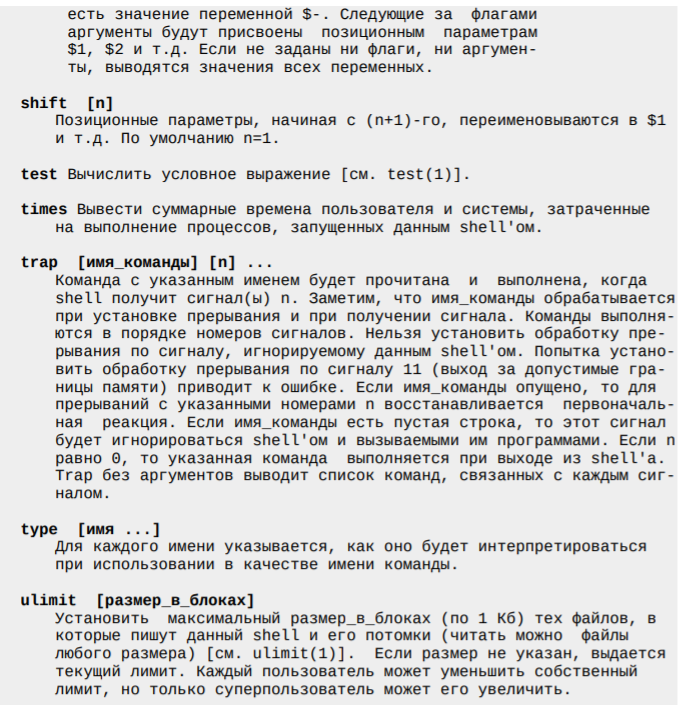


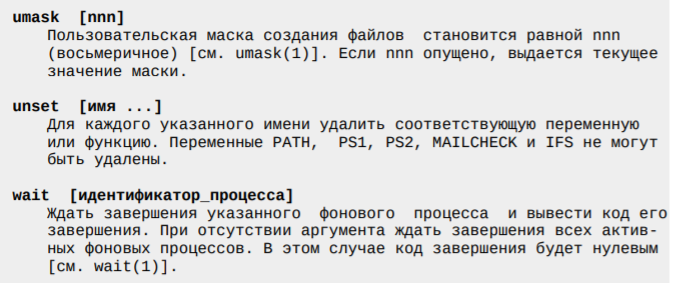












### **29. Фоновый и приоритетный режимы.**

В интерактивном режиме, shell взаимодействует посредством консоли :

* • пользователь в консоли набирает цепочку символов и, в конце цепочки нажимает клавишу «Ввод»;
* • shell проводит синтаксический анализ введенной цепочки, выделяет простые команды, формирует конвейер команд и запускает задание;
* • когда задание, которое может состоять из множества процессов, завершится, shell выдаст на консоль приглашение на ввод новой цепочки символов.

Задания, выполняющиеся указанным способом, называются заданиями, выполняющимися в ***приоритетном режиме***. Shell блокирует ввод новых цепочек символов до завершения таких заданий.

Если пользователь, перед нажатием клавиши «Ввод» укажет символ &, то задание будет выполняться в ***фоновом режиме***. В этом случае:

* • shell выводит на консоль номер задания, заключенный в квадратные скобки, и номер PID родительского процесса задания;
* • после этого, shell выводит на консоль приглашение пользователю для ввода новой цепочки символов.

Для перевода ***фонового*** задания в ***приоритетный*** режим работы, используется команда fg (foreground).

### 

### **30. Отмена заданий.**

Для ***отмены заданий***, выполняющихся в фоновом режиме, используется команда kill, которая в качестве аргумента может использовать номер задания или PID. В результате применения этой команды, задание прекращает работу, а созданные им процессы уничтожаются.

### **31. Прерывания.**

Выполнение задания в приоритетном режиме можно ***прервать***, используя комбинацию клавиш Ctrl-Z. При этом:

* выполнение задания приостанавливается и shell выдает пользователю приглашение на ввод новой цепочки символов;
* командой fg (foreground) задание можно перевести в приоритетный режим;
* командой bg (background) задание можно перевести в фоновый режим.

### **32. Завершение работы ОС.**

Если запустить ОС может любой пользователь, который включит питание ЭВМ и, возможно, выберет в меню тип загружаемой ОС, то для выключения компьютера, пользователь должен иметь права на запуск команд:

* halt [OPTION] …
* poweoff [OPTION] …
* reboot [OPTION] …
* shutdown [OPTION] … TIME [MASSAGE]

Работая в графической оболочке, пользователь для выключения ЭВМ использует соответствующее меню. В этом случае, команды и сам процесс выключения ОС — скрыт от пользователя.

### **33. Устройства компьютера (УК).**

С одной стороны под термином УК ассоциируются с аппаратурой ЭВМ, с другой УК являются ресурсы ОС, которые управляются ядром ОС, с третьей сами файловые системы ОС располагаются на УК.

Указанное выше противоречие, заключающееся в том, что устройства отображаются в ФС, которые сами находятся на устройствах, разрешается следующим образом:

• все устройства ОС делятся на блочные и символьные;

• блочные устройства ОС могут содержать файловые системы;

• символьные устройства ОС — устройства ОС, не являющиеся блочными;

• в ядре ОС создаются блочные устройства, которые не относятся к аппаратным средствам ЭВМ: псевдоустройства ядра ОС или устройства nodev;

• псевдоустройства ядра ОС имеют имена, которые почти все совпадают с

именами соответствующих файловых систем; эти имена соответствуют вершинам файловых систем (ФС);

• псевдоустройство с именем rootfs является корнем виртуальной файловой

системы (VFS — Virtual File System).

Все псевдоустройства ядра ОС обозначены как nodev.

Все ОС UNIX и Linux отображают имена файлов устройств в директорию /dev корневой файловой системы. Для устройств, точкой монтирования ядра ОС является udev.

В терминологии ядра, имя устройства, которое отображается в директории /dev,

называется узел (node).

NODE - узел - специальная именованная структура.

Узлы создаются как для блочных, так и для символьных устройств.

Каждый узел имеет имя, тип, старший\_номер и младший\_номер.

Все узлы хранятся в специальной таблице (специальной файловой системе).

Для создания устройств используется утилита mknod, в формате;

mknod [OPTION]... NAME TYPE MAJOR MINOR

где NAME — имя устройства;

TYPE равно b для блочного устройства;

TYPE равно c для символьного устройства;

TYPE равно p для канала FIFO;

MAJOR, MINOR — старший и младший номер устройства

Пример: [ -e /dev/null ] || mknod /dev/null c 1

Для создания узлов необходимо знать не только имена и типы устройств, но и их номера (младшие и старшие). Наличие большого количества различных физических устройств ЭВМ и их различная группировка, создает ряд известных проблем, при создании и сопровождении узлов.

### **34. BOOT-сектор и разделы винчестера.**

На одном из разделов винчестера размещается ФС, рассматриваемая ядром ОС как корневая ФС.

Все блочные устройства, на которых размещаются конкретные ФС имеют свою

структуру, определяемую физическим типом устройства. Данную структуру имеют физ. устройства - винчестеры.

Так как его часто использовали, то ввели стандартизацию. По стандарту:

• классическая структура MBR;

• новая структура GPT

Широкое применение устройства типа винчестеров привело к переносу данной структуры на flashUSB, оно позволяет использовать его как блочное устройство.

Ограничиваясь только классической структурой MBR, можно выделить:

• главный загрузочный сектор (MBR), который не зависит от типа ОС;

• загрузочные секторы (блоки) логических дисков (разделов), которые зависят от ОС только в плане поддерживаемых ей типов ФС;

• специальные области разметки и корневой каталог, зависящие от типа файловой системы;

• область данных – файлы и каталоги конкретной файловой системы;

• цилиндр для выполнения диагностических операций чтения-записи.

Для поддержки такой структуры блочных устройств, каждая ОС имеет специальные системные утилиты.

В ОС УПК АСУ:  
 fdisk - позволяет показать все блочные устройства ЭВМ, а также обеспечить разбиение устройства на нужное количество разделов

mount — позволяет смонтировать (подключить) конкретную ФС

umount — позволяет размонтировать (отключить) конкретную ФС

Задача загрузки первых реализаций ОС сводилась

• к использованию загрузочного блока для загрузки ядра и передачи ему информации о разделе конкретного блочного устройства;

• ядро ОС, проведя инициализацию внутренних параметров и внешних устройств ЭВМ, находила этот раздел и монтировала его как корневую файловую систему.

Хотя каждая ФС была оптимизирована под ядро конкретной ОС, подобный подход имел две существенные проблемы:

• правильное монтирование корневой ФС было возможным только после предварительной инсталляции ОС, посредством специального инсталлятора;

• перенос данных с других («не родных») ФС был крайне затруднительным.

### **35. Загрузочные сектора разделов.**

Для блочного устройства типа винчестер, минимальным физически читаемым объемом информации является сектор, в свою очередь сектор равен 512 байт в современных ЭВМ.

Понятие сектора связано физической организацией обмена данными в блочных устройствах.

Понятие блока связано с логической организацией обмена данными между ОС и блочными устройствами.Размер блока может быть равен сектору, либо кратен целому числу секторов.

Понятие блока вводилось из-за:

Преодоления ограничений, которые накладываются способом адресации разделов в таблице MBR и ПО BIOS, которые ограничивали возможный объем используемых блочных устройств.

Кластер — это последовательный набор секторов винчестера, который на логическом уровне рассматривается как единица обмена данными между ОС и блочным устройством.

Конструктивные особенности блочных устройств типа винчестера связаны с тем, что линейная скорость движения головки чтения/записи по внешним трекам диска выше, чем по внутренним. Соответственно, на внешних треках можно расположить больше секторов, чем на внутреннем, что позволяет более эффективно использовать поверхность диска, но делает непригодной систему адресации секторов: (CHS).

Для решения этой проблемы было принято использовать LBA. LBA — стандартизованный механизм адресации и доступа к блоку данных на блочном устройстве, при котором системному контроллеру нет необходимости учитывать специфику накопителя.

LBA перекладывает адресацию секторов на контроллер винчестера, а для ОС предоставляется упорядоченная последовательность логических блоков. Данная структура блочных устройств GPT снимает проблему логической адресации больших объемов устройств, расширив формат указателей адреса и стандартизировать размер LBA на уровне одного сектора (512 байт).

Для загрузки ОС необходимо в разделе корневой файловой системы выделить область для ПО загрузчика, а в программный код первых 446 байт MBR прописать адрес загрузчика. Очевидно, что такой адрес должен указывать на фиксированное местоположение ПО загрузчика относительно начала раздела. Поэтому, практически все ФС включают в свою структуру некоторый блок загрузки. Как следствие, такой подход позволяет загружать только одну ОС с одного блочного устройства.

Альтернативный подход предполагает смещение начала первого раздела относительно сектора MBR. Такое смещение позволяет записать сразу же после сектора MBR ПО загрузчика, что и используется нами при установке ПО GRUB.

### **36. Структура файловой системы FAT32 (VFAT).**

**FAT32** - файловая система компании Microsoft. Она является преемницей файловых систем FAT8, FAT12 и FAT16.

**VFAT** — расширение FAT, появившееся в MS Windows 95 и дающее возможность использовать имена файлов длиной до 255 символов, в кодировке UTF-16LE. Фактически, VFAT и используется во всех ОС, как универсальный хранитель информации.

**FAT** – File Allocation Table или Таблица размещения файлов. 32 – число бит, используемое для нумерации кластеров (блоков) раздела файловой системы (на самом деле используется 28 бит).

**Кластер** – это объем данных, которыми оперирует файловая система. Обычно, для 8 Гбайт раздела используется 4 КБайтные кластеры (8 секторов диска).

**Один сектор диска** – 512 байт

Загрузочный сектор раздела содержит:

блок параметров диска (BPB), в котором содержится информация о разделе: размер и количество секторов, размер кластера, метка тома и другие;

загрузочный код – программу, с которой начинается процесс загрузки операционной системы

На этапе логического форматирования каждого раздела (логического диска) создаются четыре логических области:

• загрузочный сектор (boot sector);

• таблица размещения файлов (FAT1 и FAT2);

• каталог;

• область данных.

Загрузочный сектор содержит в себе, кроме кода загрузчика, таблицу BPB и двухбайтовую сигнатуру

BPB — BIOS Parameter Block — таблица содержащая множество параметров, определяющих характеристики блочного устройства и самой файловой системы.

Резервная загрузочная запись, как правило, располагается в секторах 7-9 раздела.

Таблица размещения файлов (FAT) – это массив целых чисел с длинной, равной количеству кластеров раздела файловой системы.

Номер элемента этого массива – это номер кластера в разделе файловой системы. Отдельный элемент массива:

• для FAT16 — это 2-х байтовые числа;

• для FAT32 — это 4-х байтовые числа.

Значение элемента таблицы FAT — это ссылка на следующий номер элемента таблицы FAT.

За таблицами FAT располагается корневой каталог файловой системы

Корневой каталог файловой системы – список записей по 32 байта.

Система FAT содержит два типа имен:

• короткие имена — до восьми байт имя и три байта расширение имени;

• длинные имена — до 255 символов в формате UTF-16LE.

Формат записи короткого имени файла помещается в одну 32-байтовую запись

Для длинных имен файлов, в формате Unicode-16, используется несколько 32- байтовых записей каталога в специальном формате.

Область данных — все остальные кластеры раздела.

Они используются для хранения подкаталогов и файлов.

Подкаталоги – это файлы, содержащие последовательности 32-битных записей имен файлов и подкаталогов, указанного выше формата.

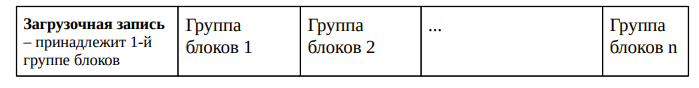
Содержимое файлов — данные некоторой последовательности кластеров. Последовательность кластеров — произвольная и неупорядоченная, но отражена в виде цепочки, в таблице FAT.

### **37. Структура файловой системы EXT2FS.**

ext2 (ext2fs) - Second Extended File System — базовая файловая система ОС Linux

Поддерживает файлы размером 2 ГБайт — 2 ТБайт.

Размер файловой системы - 2 — 32 ТБайт.

Структура дискового раздела ext2fs представляет последовательность группы блоков

Каждая группа блоков состоит из последовательности блоков, которые также нумеруются, начиная с 1.

Каждая группа блоков имеет одинаковое число блоков, кроме последней, которая может быть неполной.

Начало каждой группы блоков имеет адрес, который может быть получен как:

Address = (номер\_группы — 1)\* (число\_блоков\_в\_группе)

Размер блока может быть 1, 2 или 4 килобайта, что определяется в момент форматирования – при создании файловой системы.

Блок является адресуемой единицей дискового пространства.

Каждая группа блоков имеет одинаковое строение:

### **38. Сравнение файловых систем.**

Два подхода к обработке данных:

• простоту управления, характерную для FAT32;

• скорость обработки больших файлов, характерную для ext2fs.

Общая метаструктура:

• секторов загрузчика ОС;

• суперблока, характеризующего файловую систему и раздел в котором она расположена;

• набора таблиц для разметки и учета использованных блоков данных;

• корневого каталога, с которого начинается логическое построение ФС;

• набора блоков данных, в которых размещаются подкаталоги и файлы ФС.

Все известные файловые системы отличаются только способом реализации этих элементов метаструктуры.

**Преимущества FAT32:**

• простота реализации и эффективность использования всего физического пространства блочных устройств малой емкости;

• широкая известность и распространенность, делающая ее «универсальным хранилищем» временных ФС.

**Недостатки FAT32:**

• практическая ограниченность размера ФС и слабая защищенность, делающая ее непригодной для современных ОС и технологий хранения информации;

• высокая фрагментированность ФС в процессе эксплуатации, что снижает скорость ее работы и негативно воздействует на устройства хранения данных.

**Преимущества ext2:**

• высокое быстродействие;

• поддержка файлов до 2 Гбайт.

**Недостатки ext2**:

• отсутствие журналирования, снижающее ее надежность;

• недостаточные, по современным меркам, размеры поддерживаемых разделов ФС и размера файлов.

Журналируемая файловая система — это ФС, которая сохраняет список изменений, проводимых с файловой системой, перед фактическим их осуществлением.

Записи будущих фактических изменений хранятся в отдельной части ФС, называемой журналом (journal) или логом (log).

### **39. Стандартизация структуры ФС.**

Структуру ФС как организованный набор файлов и директорий, видимый программами (процессами) пользовательского режима работы отдельных ОС.

В общем случае, структура ФС на уровне файлов и каталогов может быть произвольной, за некоторыми ограничениями, например:

• обязательно наличие корневой ФС ОС;

• обязательно наличие корневого каталога отдельных ФС.

/

/dev /usr /home /bin /…

/usr/local /usr/man

Хорошо заметно, что ФС структурирована

• dev — директория размещения узлов устройств ЭВМ (devices);

• usr — директория ориентированная на универсальные ресурсы ОС;

• home — директория для рабочих (домашних) областей пользователей;

• bin — директория расположения исполняемых файлов ОС (binary).

В рамках проекта GNU, стал разрабатываться стандарт на структуру файловой системы (Filesystem Standard), который был выпущен в феврале 1994 года и получил обозначение FSSTND.

В общем случае, файловая система Linux разделена на три крупных уровня иерархий. **Первый уровень иерархии ФС** составляют каталоги, которые управляются пользователем root и другими администраторами системы.

**Второй уровень** **иерархии ФС,** по замыслу, содержит данные, которые доступны только для чтения и могут использоваться всеми пользователями или несколькими компьютерами сети. В идеале, модифицироваться могут только файлы каталога /usr/local, в который администратор может устанавливать новые программы, не поставляющиеся с системой.

**Третий уровень иерархии ФС** — подобен второму и продолжается в директории /usr/local.

Управление видимостью дерева ФС осуществляется с помощью утилиты chroot. В частности, она используется во время выполнения процедуры login. С другой стороны, видимая для пользователя ФС, создает для него иерархическую систему координат, в пределах которой он принципиально может адресовать любой файл. Следующий аспект работы пользователя состоит в том, что он всегда находится в некоторой точке ФС, которая связана с некоторой ее директорией.

В какой директории находишься pwd. Для перемещения используется cd.

Наиболее распространенные команды для работы с директориями:

• mkdir директория; - создание директории;

• rmdir директория; - удаление директории;

• ls [опции] [шаблон]; - вывод на консоль списка файлов и директорий;

• cat файл; - вывод содержимого файла на консоль.

### **40. Модули и драйверы ОС.**

Ядро ОС — это большая программа, работающая в собственном защищенном (привелигированном) режиме. Любой процесс имеет доступ к устройствам только через ядро ОС т.к. работает в режиме пользователя.

Чтобы работать с конкретным устройством, ядро ОС имеет специальное ПО драйвер. ПО драйвера имеет два интерфейса:

• интерфейс вызова драйвера ядром ОС, который использует концепцию узла ФС или node, подразделяя устройства на символьные и блочные;

• интерфейс устройства, который использует сам драйвер для работы с конкретным устройством через общую шину ЭВМ.

Принципиальная разница между этими интерфейсами:

• интерфейс вызова драйвера ядром ОС определяется разработчиками ОС и различается от ее типа: разный интерфейс для разных ОС

• интерфейс устройства определяется архитектурой аппаратной части ЭВМ и конструктивными особенностями конкретного устройства.

Учитывая большое разнообразие архитектур аппаратной части ЭВМ, драйверы традиционно пишутся в виде модулей, которые хранятся в ФС ОС и загружаются по мере необходимости.

**Модуль** — системное ПО ОС, которое не обязательно является драйвером. Главное, что это ПО предназначено для работы в защищенном режиме пространства ядра ОС.

Использование модулей предполагает учет их особенностей:

• управление модулями из режима пользователя осуществляется утилитами insmod, rmmod, lsmod, modinfo и modprobe;

• разработка модулей предполагает знание не только технологии их написания и отладки, но также - архитектуры и ПО ядра ОС;

• вывод информации в пространство пользователя модуль осуществляет посредством функции printk(...), которая записывает сообщение в файл /var/log/ messages;

• для просмотра сообщений модулей ядра используется утилита dmesg;

• файлы конфигурации и сами модули находятся в директориях ФС: /etc и /lib/modules.

### **41. Системные вызовы ОС по управлению файловыми системами.**

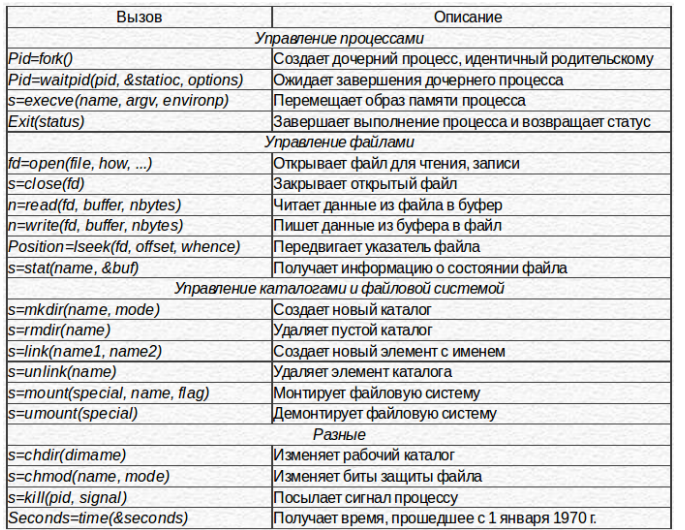
Системных вызовы ОС, разделенных на группы:

• управление процессами;

• управление файлами;

• управление каталогами и файловыми системами;

• разные, которые нельзя полностью отнести к предыдущим группам.



### **42. Три концепции работы с устройствами.**

Три концепции (технологии) именования и отслеживания актуальности наличия устройств: **manual**, **udev** и **systemd**.

MANUAL

Концепция manual (ручная) появилась первой. Она предполагает использование системного вызова mknod(...) или утилиты mknod. Обычно, созданием устройств занимается программа-инсталлятор ОС, которая проверяет архитектуру компьютера на наличие имеющихся устройств, а также создает узлы устройств: автоматически или по требованию администратора ОС. Первоначально, именование и параметры узлов придумывались разработчиками ОС и были достаточно разнообразны. Со временем, стала проводиться стандартизация, учитывающая известные типы оборудования. Например:

• почти все ОС имеют устройство консоли /dev/console;

• разделы винчестеров, с контроллерам SATA или SCSI, стали именоваться - /dev/sda1, /dev/sda2, …, /dev/sdb1, /dev/sdb2, ...

Окончательно, процесс стандартизации завершился созданием скрипта shell — MAKEDEV, который упорядочил использование имен и других параметров узлов. Часто, найти этот скрипт можно в директории /dev или /sbin.

UDEV

С начала 2000-х годов, в технологиях ОС, стала применяться файловая система devfs. Цель проекта - решить основную проблему ОС UNIX/Linux, связанную как с катастрофически возрастающим объемом директории /dev, так и с поддержкой технологии PnP. Данный проект предполагал использование специального демона devfsd, который периодически обращался к ядру ОС с целью контроля за состоянием имеющегося оборудования:

• при обнаружении подключения нового оборудования, этот демон создавал соответствующий узел (inode) в директории /dev;

• в случае обнаружения ситуации, что оборудование отключилось, он соответственно удалял узел из директории /dev.

Существенным недостатком devfs является неэффективное расходование ресурсов ЭВМ, связанное с его постоянным циклическим запуском, поэтому современные системы ОС UNIX/Linux перешли на более новую технологию udev, используя для директории /dev файловую систему devtmpfs

udev решила проблему Plug and Play

При старте ОС запускается демон udevd, который, используя шину D-Bus, отслеживает подключение новых устройств и создает для них нужные узлы. Когда устройства отключается от ЭВМ, udevd удаляет соответствующие им узлы, кроме тех, которые были созданы в ручную. Одновременно, также решается проблема поддержания в активном состоянии не использованных узлов.

SYSTEMD

Ее цель — унификация взаимодействия между процессами режима пользователя. В частности:

• отслеживание динамики подключения устройств (вместо udevd);

• эффективное отслеживание групп процессов, порожденных системными вызовами fork(...) и объединенных одной прикладной целью (cgroups).

### **43. Разделы дисков и работа с ними.**

При условии, что все ФС созданы работать с разделами дисками можно с помощью:

• необходимо определить точки монтирования (директорий) корневой ФС ОС для подключения к ним (монтирования) нужных сопутствующих ФС и затем, правильно записать желаемую конфигурацию в файле /etc/fstab;

• выполнить команду mount -a, и желаемая структура ФС ОС будет создана для последующей эксплуатации.

Данный подход требует определенной квалификации и работает если ФС стабильна и не требует обновлений

Если же необходимо провести модернизацию связанную с:

• динамическим подключением других («не родных») ФС;

• подключением к сетевым ресурсам данных

Решением проблемы является разработка и использование необходимых модулей ядра ОС. Но поскольку структура ФС достаточно сложна, а модули подключаются к ядру ОС, то:

• размер ядра ОС значительно увеличивается;

• время разработки и последующей отладки ПО модулей становится неприемлемо большим.

Также существует виртуальная файловая система VFS:

Суть идеи — расположить VFS между приложениями и конкретными файловыми системами на внешних носителях.

Это позволяет сконцентрировать в одном месте пользовательский интерфейс для разных типов ФС, что создает унифицированный эталон для разработки драйверов ОС.

VFS:

• упрощает процесс создания драйверов ФС, что повышает их надежность;

• сокращает время разработки, что создает конкурентные преимущества самой ОС;

• уменьшает время отладки и исправления ошибок, что повышает актуальность их применения.

Основу управления ФС составляют утилиты mount и umount.

### **44. Монтирование и демонтирование устройств.**

**Монтирование файловой системы** — подключение конкретного раздела внешнего блочного устройства к конкретному каталогу ФС

**Демонтирование файловой системы** — отключение конкретного раздела внешнего блочного устройства от конкретного каталога ФС

Общее правило монтирования/демонтирования ФС:

• **монтирование осуществляется командой mount** с указанием узла ФС и директории монтирования, при этом: старое содержимое директории становится невидимым, а монтируемая ФС — видимой;

• **демонтирование осуществляется командой umoun**t с указанием или узла ФС или директории монтирования, при этом: старое содержимое директории становится видимым, а демонтируемая ФС — невидимой;

Пример использования mount -t Тип Устройство Директория;

mount -t ext2 /dev/sda2 /mnt

umount /dev/sda2;

Команда mount, без параметров, выдает список монтированных в данный момент устройств.

Для централизованного управления монтированием ФС, используется файл /etc/fstab. Структура файла имеет вид:

Устройство Каталог Тип Опции fs\_freq fs\_\_passno

где fs\_freq — определяет, должна ли создаваться резервная копия этого раздела с помощью команды dump; значение 0 отменяет dump;

fs\_\_passno — указывает в какую очередь данная ФС проверяется на целостность, при запуске Linux: 0 — проверка не требуется; 1 — для корневой ФС; 2 — для других ФС

### 

### **45. Файловые системы loopback, squashfs, overlayfs и fuse.**

Термин **loopback** означает - «обратная петля». Применительно к нашей тематике он будет означать:

• циклическое устройство, когда мы будем говорить о блочном устройстве;

• циклическая ФС, когда мы будем говорить о файловой системе, которая создана в отдельном файле. Узлы циклических устройств находятся в директории /dev.

Циклические ФС обычно создаются в файлах, которые могут быть сжатыми и зашифрованными

Широкое распространение получила технология хранения файловых систем в сильно сжатых файлах для live-дистрибутивов. Такой подход позволяет практически на прямую использовать технологии созданные для работы с ФС на CD дисках. Например, если устройство /dev/sda1 содержит файловую систему ntfs, в которой имеется файл /asu64upk/upkasu/usrfs.sfs, то монтирование такой ФС (типа squashfs) к директории /usr осуществляется двумя командами

Другая специальная ФС - overlayfs — является вариантом реализации каскадной файловой системы для ОС Ubuntu и ее клонов.

Каскадная файловая система (КФС) — это виртуальная ФС, позволяющая «прозрачно видеть» две изолированные ФС как одну.

Каскадно-объединенное монтирование — одновременное монтирование разных нескольких файловых систем как одну, например для объединения ФС нескольких сайтов.

FUSE — Filesystem in Userspace — файловая система в пространстве пользователя. FUSE реализована в виде модуля для UNIX-подобных ОС. Этот модуль позволяет пользователям без привилегий создавать свои ФС, без необходимости переписывать код ядра ОС. Это достигается за счет запуска кода файловой системы в пространстве пользователя, в то время как модуль FUSE предоставляет мост для актуальных интерфейсов ядра.

### **46. Дисковые квоты.**

Рано или поздно может случиться так, что некоторая ФС на внешнем носителе,

подключенная к древовидной структуре ядра ОС, заполнится и блокирует нормальную работу ОС.

По умолчанию, ФС ext2fs резервирует 5% места для пользователя root.

Принято считать, что для нормальной работы ФС требуется свободное пространство:

• не менее 10% дискового пространства (пространство раздела);

• не менее наибольшего файла в файловой системе.

Рекомендуется размешать /home, /opt, /tmp, /var, /usr/local в разных разделах диска.

Определить свободное пространство диска можно командой df.

Администратор ОС должен отслеживать наличие достаточного дискового пространства ФС. Для этих целей имеется специальное системное ПО под общим

названием quota. Естественно, ядро ОС должно поддерживать команды этого

пакета.

Чтобы включить квоты для конкретной файловой системы, нужно в корень ФС

поместить бинарные файлы:

• quota.user — для персональных квот;

• quota.group — для групповых квот.

Управление квотами выполняется с помощью команд:

quotastats Проверка поддержки квот ядром ОС.

quotacheck Сканирование заданной ФС и первоначальное создание файлов

quota.user и quota.group.

edquota Редактор квот.

quotaon Активация настроек квот для ФС.

quotaoff Деактивация настроек квот для ФС.

quota Проверка пользователем, установленных для него квот

### **47. Однопользовательский и многопользовательский режимы работы ОС.**

Современное понятие однопользовательского режима означает не тот факт, что ОС не может поддерживать многопользовательский режим, а то что:

• или отключен контроль разграничения прав пользователей, при одновременной изоляции ЭВМ от внешних воздействий, например, отключение от сети;

• или остановлена работа программ всех пользователей, кроме администратора, например, суперпользователя root.

Например, современные ОС загружаются в два этапа:

• на первом этапе, после загрузки и запуска ядра ОС специальным загрузчиком, например GRUB, ядро распаковывает в оперативную память временную файловую систему и запускает первый процесс init;

• сам процесс init — обычно скрипт, выполняемый интерпретатором shell, устанавливает необходимые модули ОС, ищет и монтирует корневую файловую систему, создает терминальные устройства и запускает на них программы login, удаляет временную файловую систему и завершает работу; все это делается в однопользовательском режиме ОС;

• на втором этапе, пользователи, которые начинают проходить процедуру login, работают уже в многопользовательском режиме ОС.

Информационное обеспечение многопользовательского режима ОС, прежде всего, поддерживается группой системных файлов.

Файл /etc/passwd, каждая строка которого имеет формат: username:password:UID:GID:GEOS:homedir:shell

Учитывая большую важность этой информации, содержимое файла /etc/passwd

дублируется в файл /etc/passwd-.

Информация о группах пользователей и дубль этой информации хранятся в файлах

/etc/group и /etc/group-, в формате:

groupname:password:GID:userlist

### **48. Разграничение прав пользователей.**

Общая парадигма концепции пользователя подразумевает, что все пользователи ОС

работают автономно и не мешают друг другу, кроме системного администратора.

Это достигается двумя основными мерами:

• каждый пользователь имеет право работать только с теми файлами, директориями и файловыми системами, к которым он имеет доступ;

• пользователь root, с идентификатором UID=0, может делать абсолютно все.

Главным показателем пользователя является его идентификатор UID: с увеличением номера UID права пользователя уменьшаются.

Аналогичный критерий справедлив для групп пользователей, права которых определяются идентификатором GID.

Условно, все пользователи разделяются на две категории:

• системные пользователи — root, sysadm и другие администраторы;

• обычные пользователи — те, которые используют прикладное программное

обеспечение ОС и не занимаются администрированием.

Условность такого разделения подтверждается тем фактом, что в первых ОС идентификаторы обычных пользователей начинались с номера 100.

Со временем, разработчики прикладного ПО стали столь интенсивно использовать

идентификаторы, что было принято решение:

• системные пользователи — UID < 999;

• пользователь live-дистрибутива — UID=999 и GID=999;

• обычные пользователи — UID > 999.

Обычному пользователю доступны:

• все файлы и каталоги его домашней директории, положение которой задано системной переменной HOME;

• права записи в каталоги /tmp и /var/tmp;

• права монтирования и демонтирования внешних устройств, которые прописаны в файле /etc/fstab с опцией user.

Временная смена прав доступа на права другого пользователя достигается командой: su [-] [username]

Смена прав пользователя ОС связана с его действиями в системе.

Действия пользователя в системе определяется работой программ (процессов),

которые пользователь запускает.

Действительные (реальные) ID пользователя и ID группы — это числовые (двухбайтовые) значения UID и GID, записанные в файлах /etc/passwd и /etc/group во

время создания пользователя в системе.

Эффективные ID пользователя и ID группы — это числовые (двухбайтовые) значения, которые учитываются в системе при выполнении конкретного процесса:

• дочерний процесс, создаваемый системным вызовом fork(…), получает эффективные ID от своего родителя;

• процесс, модифицируемый одним из системных вызовов exec(…), устанавливает эффективные ID в зависимости от значений битов SUID и SGID, присутствующих в поле i\_mode индексного дескриптора файла: если биты SUID и SGID — установлены, то эффективные ID берутся из дескриптора файла, а если нет, то устанавливаются в значения действительных (реальных) ID пользователя, запустившего процесс.

Сохраненные ID пользователя и ID группы — это числовые (двухбайтовые) значения первоначальных эффективных ID, которые сохранены в памяти процесса с

помощью системных вызовов getuid(...) и getgid(...), сразу же после завершения

системного вызова exec(...).

Установка новых значений эффектиных ID выполняется с помощью системных вызовов setuid(...) и setgid(...).

### **49. Login и система доступа Linux-PAM.**

Любому пользователю, для нормальной работы с ОС, необходимо пройти процедуру регистрации в системе: получить имена, идентификаторы, пароли и место для работы. Указанную процедуру выполняет администратор ОС. В результате регистрации:

• имена, идентификаторы и пароли будут записаны в соответствующие файлы: /etc/passwd, /etc/group, /etc/shadow и /etc/gshadow;

• в директории /home будет создана директория с именем пользователя;

• в директорию /home будут перенесены директории и файлы, находящиеся в директории /etc/skel, которые составляют начальный скелет рабочей области любого пользователя.

Процедура входа в ОС может быть:

текстовой, когда на консоль терминала выводится приглашение login:,

или графической, когда выводится некоторое стилизованное окно приглашения. В любом случае, требуется набрать имя и пароль, а возможно и другие сведения, например, домен или язык работы с системой. Это зависит от настроек «Менеджера сеансов». После ввода необходимой информации начинается с процедуры login, которая подразделяется на:

• идентификацию (аутентификацию), подразумевающую совпадение имени и пароля, зарегистрированных в системе;

• авторизацию, подразумевающую создание среды для работы программ пользователя и фиксирование прав, которыми пользователь обладает.

Фактически, авторизация не заканчивается завершением утилиты login. Она проводится постоянно, когда пользователь обращается к файлам или взаимодействует с процессами

Поскольку методы авторизации могут быть различны, а пользователю даже приходится обращаться к программам, требующим смены пользователя, то смена парадигмы обеспечения безопасности работы ОС, приводит к перезаписи большого количества системного ПО.

Системы PAM (Pluggable Authentication Modules) — заменяемые модули идентификации

Система PAM введена для создания дополнительного уровня защиты между приложениями и различными протоколами и способами идентификации и авторизации. Модули PAM — это динамически загружаемые библиотеки, которые находятся в директориях /lib/security или /usr/lib/security.

Все приложения используют универсальный интерфейс, PAM API, а уже модули PAM выбирают стратегию поведения и протоколы согласно файлам конфигурации: /etc/pam.conf либо /etc/pam.d/...

### **50. Команды управления пользователями.**

**В зависимости от контекста**, термин пользователя понимается как:

• человек, работающий за терминалом компьютера;

• концептуальная основа выделения владельцев, группы и других, связанные с концепциями файла и процесса;

• система обозначений, используемая имена и числовые идентификаторы, распространяемая на все ПО ЭВМ;

• система разграничений, ограничивающая и специализирующаяся права использования ПО ЭВМ.

**Хотя работа с пользователями** предполагает всего три действия: **создание, удаление и модификацию**, - наличие множества конфигурационных файлов, привязанных к древовидной структуре ФС, превращает сам процесс управления в непростую задачу.

Чтобы упростить этот процесс, используются три команды (утилиты): **useradd, userdel и usermod**, расположенные обычно в директории /usr/sbin.

Особое место в управлении пользователями занимает утилита **passwd**, которая управляет паролями пользователя и ограничивает его работу на уровне сеанса.

### **51. Подсистема управления процессами.**

**Подсистема управления процессами находится в ядре ОС**. Основная ее функция - обеспечение мультипрограммного режима работы ОС, что связано с:

• созданием процессов в системе и удаление их из системы, что предполагает управление основной памятью ЭВМ;

• переключением процессов в режимы «Готовность», «Выполнение» и «Ожидание».

Качественное выполнение этой функции требует планирования подсистемой своих действий, что, в общем случае, не является однозначно решаемой задачей.

**Чтобы оценить сложность** решаемой задачи планирования, рассмотрим перечень требований предъявляемых к ней, в зависимости от целевых аспектов различных прикладных систем.

**Все системы** должны обеспечить:

• Справедливость — предоставление каждому процессу справедливой доли процессорного времени.

• Принудительное применение политики - контроль за выполнением принятой политики;

• Баланс — поддержка занятости всей системы.

**Системам пакетной обработки данных** необходима:

• Пропускная способность — максимальное количество задач в час.

• Оборотное время — минимизация времени, затрачиваемого на ожидание обслуживания и обработку задачи.

• Использование процессора — поддержка постоянной занятости процессора.

**Интерактивным системам** важно:

• Время отклика — быстрая реакция на запросы.

• Соразмерность — выполнение пожеланий пользователя.

**Системам реального времени** требуется:

• Окончание работы к сроку — предотвращение потери данных.

• Предсказуемость - предотвращение деградации качества в мультимедийных системах

Перечисленные выше требования к различным системам показывают, что можно построить большое число алгоритмов планирования, но все они окажутся обоснованными только в ограниченных условиях. Для примера, рассмотрим некоторые из них.

**Планирование** в **системах пакетной обработки** данных:

• Первым пришел — первым обслужен. Является наиболее простым алгоримом планирования, который выделяет первому процессу, запросившему процессор, все время, необходимое для его завершения.

• Кратчайшая задача — первая. Позволяет очень быстро выполнять маленькие задачи, но требует знания времени их выполнения.

• Наименьшее оставшееся время выполнения. Если имеется задача, время выполнения которой меньше, чем время завершения текущей, то текущая задача останавливается, а минимальная, по времени исполнения, запускается. Здесь требуется также знать время выпонения процессов.

• Трехуровневое планирование. Здесь имеется впускной планировщик, который выбирает задачи из общей очереди и передает их процессору на выполнение. Возможны разные варианты, учитывающие возможности процессора и устройств ввода-вывода. Второй уровень планирования определяет: какие процессы можно хранить в памяти, а какие — на диске. Этим занимается планировщик памяти.

**Планирование** в **системах интерактивной обработки**. Наиболее известны два алгоритма:

• Циклическое планирование, когда каждому запускаемому процессу выделяется квант времени, по истечении которого или по запросу устройств ввода-вывода процесс останавливается и помещается в конец очереди.

• Приоритетное планирование, когда каждому запускаемому процессу присваивается приоритет и управление передается готовому к работе процессу, с наивысшим приоритетом.

**Планирование** в **системах реального времени**, которые подразделяются на:

• Жесткие системы реального времени, требующие жестких сроков реакции на запросы каждой задачи.

• Мягкие системы реального времени, для которых нарушение сроков выполнения задач - нежелательно, но допустимо.

### **52. Системные вызовы ОС по управлению процессами.**

**Процесс** — это элементарный управляемый объект ОС, имеющий целочисленный идентификатор PID - Process Identification, обеспечивающий функциональное преобразование файлов (данных) с правами, которые определяются объектами пользователь.

системные вызовы ОС по управлению процессами содержат перечень наиболее важных функций, предоставляемых ядром ОС

**Сам процесс** самостоятельно способен реализовать лишь некоторый набор алгоритмов по обработке данных, которые ему доступны непосредственно: объявленные переменные, массивы, структуры и подобное.

**Максимально**, самостоятельные возможности процесса можно расширить до уровня возможностей всех библиотек, которые он использует.

Весь другой функциональный потенциал процесса обеспечивается:

• напрямую, посредством системных вызовов к ядру ОС;

• косвенно, посредством взаимодействия с другими процессами, которое также осуществляется через ядро ОС. Таким образом, управление процессом из него самого:

• осуществляется посредством системных вызовов к ядру ОС;

• определяется свойствами самого процесса в системе.

**Основные свойства** процесса:

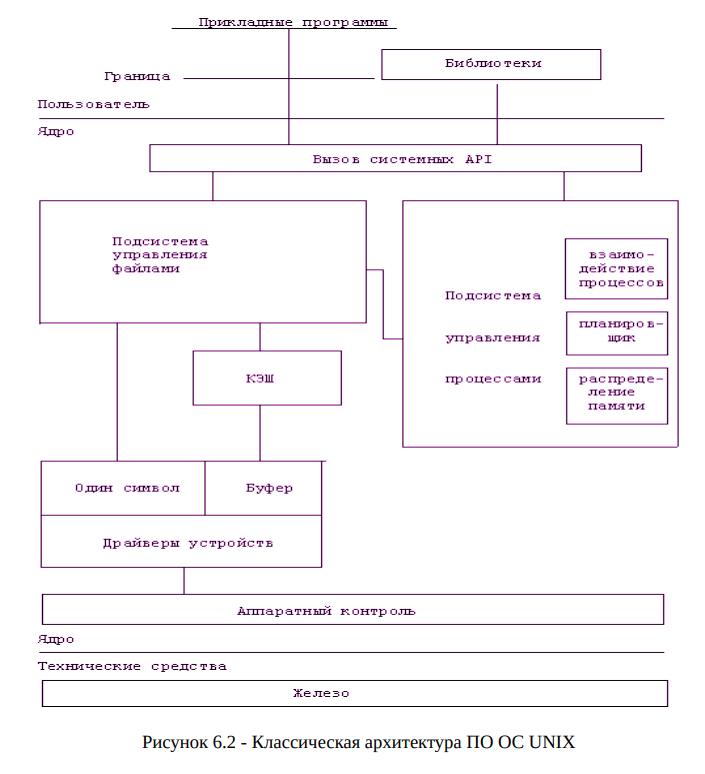
• уникальная идентификация (PID) и наличие контекста в ядре ОС;

• подверженность «жизненному циклу»: создание, функционирование и завершение;

• существование в оперативной памяти ЭВМ, выделенной для пользовательского режима работы ОС;

• сегментация структуры: сегмент кода, сегменты инициализированных и неинициализированных данных, а также по одному сегменту стека на каждую нить (thread) процесса.

**Воздействуя** на каждое из свойств процесса, можно управлять им.

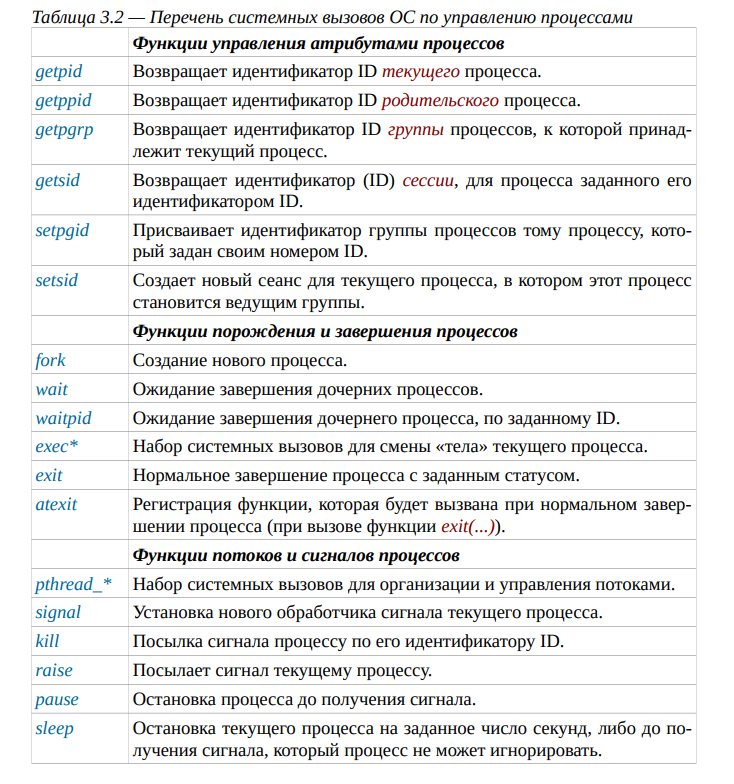


Чтобы упорядочить и, по возможности, минимизировать тупиковые ситуации между процессами, системные вызовы к ядру **ОС группируются**, как было ранее показано на рисунке 6.2, на три большие части:

• планировщик, отслеживающий контест процесса и его «жизненный цикл»;

• подсистему распределения памяти, которая в частности, обеспечивает функции для динамического выделения памяти процессам;

• подсистему взаимодействия процессов, в которую кроме сигналов включаются функции синхронизации в виде семафоров, разделяемая память и передача сообщений.



### **53. Стандарты POSIX и сигналы.**

**стандарты POSIX** отражают стремление унифицировать средства взаимодействия процессов, необходимые для адекватного отражения их в инструментальные средства языка С;

Захват и ожидание задачей ресурсов может привести к тупикам или, по крайней мере, к двум негативным следствиям:

• увеличению общего времени решения задачи;

• неэффективному использованию задачей процессора ЭВМ.

В ряде случаев, удается частично устранить эти недостатки, разбив задачу на подзадачи, которые решаются отдельными процессами ОС, или синхронизировав взаимодействие процессов с помощью средств отличных от средств их порождения и завершения.

Важнейшим таким средством являются сигналы, информирующие процессы ОС о возникновении в системе событий.

**сигналы POSIX** отражают проблематику взаимодействия процессов, которые теоретически исполняются одновременно, но реально - активен только один процесс, а остальные находятся или в состоянии «Готовность» или в состоянии «Ожидание»

Формально стандарт POSIX-2001 под сигналом понимает “механизм”, с помощью которого процесс или поток управления уведомляют о некотором событии, произошедшем в системе или подвергают воздействию этого события.

**Примерами** подобных событий могут служить аппаратные исключительные ситуации и специфические действия процессов. Термин "сигнал" используется также для обозначения самого события.

**Говорят**, что сигнал генерируется (или посылается) для процесса (потока управления), когда происходит вызвавшее его событие:

• выявлен аппаратный сбой;

• отработал таймер;

• пользователь ввел с терминала специфическую последовательность символов;

• процесс обратился к функции kill(...) и другие.

**Иногда**, по одному событию генерируются сигналы для нескольких процессов, например, для группы процессов, ассоциированных с некоторым управляющим терминалом.

**В момент** генерации сигнала определяется, посылается ли он процессу или конкретному потоку управления в процессе.

**Сигналы**, сгенерированные в результате действий, приписываемых отдельному потоку управления, таких как возникновение аппаратной исключительной ситуации, посылаются этому потоку. Сигналы, генерация которых ассоциирована с идентификатором процесса или группы процессов, а также с асинхронным событием, например, пользовательский ввод с терминала, посылаются процессу. В каждом процессе определены действия, предпринимаемые в ответ на все предусмотренные системой сигналы.

**Говорят**, что сигнал доставлен процессу, когда взято для выполнения действие, соответствующее данным процессу и сигналу.

**В интервале**, от генерации до доставки или принятия сигнал называется ждущим. **Обычно**, он невидим для приложений, однако доставку сигнала потоку управления можно заблокировать. Если действие, ассоциированное с заблокированным сигналом, отлично от игнорирования, он будет ждать разблокировки.

**У каждого** потока управления есть маска сигналов, определяющая набор блокируемых сигналов. Обычно она достается в наследство от родительского потока.

**В данном подразделе**, мы завершаем рассмотрение базовых системных вызовов управления процессами, объединенными в группу «Функции потоков и сигналов процессов», где будут изучены следующие системные вызовы:

• signal() - установка нового обработчика сигнала текущего процесса;

• kill() - посылка сигнала процессу по его идентификатору PID;

• raise() - посылка сигнала текущему процессу;

• pause() - остановка процесса до получения сигнала;

• sleep() - остановка процесса на заданное число секунд, либо до получения сигнала.

### **54. Подсистема управления оперативной памятью.**

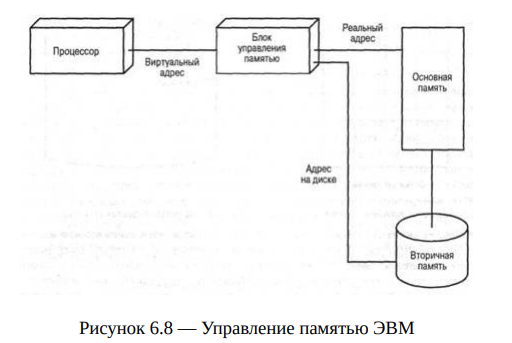
В активной форме (выполняться) процесс может только в оперативной памяти ЭВМ или ОЗУ.

**ОЗУ** — оперативное запоминающее устройство, которое часто называют основной памятью (ОП) ЭВМ.

Причина такой ситуации хорошо продемонстрирована на рисунке 6.8, где показано, что процессор:

• может выполнять только команды, находящиеся в оперативной памяти ЭВМ;

• взаимодействует с ОП через блок управления памятью (MMU).



**MMU** — Memory Managment Unit — устройство управления памятью.

**Назначение MMU** — преобразование для процессора логических адресов ОП в физические и — обратно.

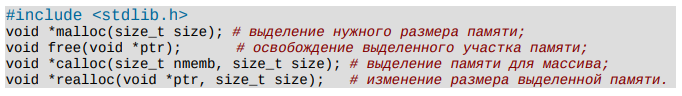
**Цель функционирования MMU** — создание для нужд процессора виртуального (относительного) пространства памяти, в котором он работает.

**Необходимость** подобной аппаратной (или программно-эмулируемой) архитектуры ЭВМ вызвана потребностью:

• реализации для ОС мультипрограммного режима работы, требующего переносимости в адресном пространстве ОП кода и данных процессов;

• использования для целей программирования языков высокого уровня, например, языка С, которые используют символьные имена переменных не привязанных к абсолютным адресам ОП.

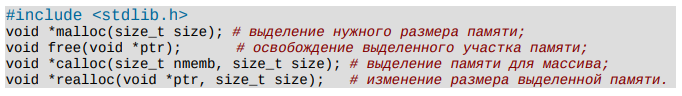
**Таким образом**, процесс может воспользоваться только четырьмя функциями, реализованными в стандарте языка С:



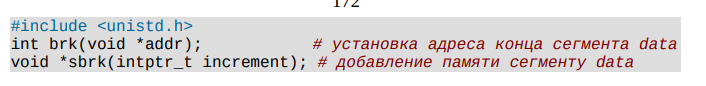
### **55. Системные вызовы ОС по управлению памятью.**

**(P.S. Не было информации в методичке)**

**Процесс** может воспользоваться только четырьмя функциями, реализованными в стандарте языка С:



Кроме того, процессу доступны системные вызовы, изменяющие адрес конца сегмента data:



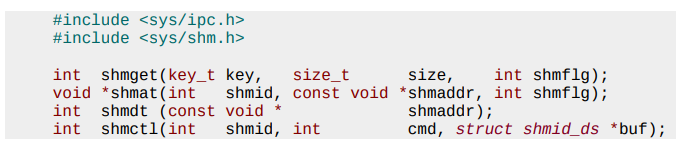
### **56. Разделяемая память.**

**В стандарте POSIX-2001** разделяемый объект памяти определяется как объект, представляющий собой память ЭВМ, который может быть параллельно отображен в адресное пространство более чем одного процесса. **Таким образом**, процессы могут иметь общие области виртуальной памяти и разделять содержащиеся в них данные.

**Единицей** разделяемой памяти является сегмент, который может быть создан одним из асинхронно взаимодействующих процессов. Такой сегмент продолжает существовать, пока один из процессов не удалит его или ядро ОС не будет перезапущено.

**Создаваемый сегмент** памяти имеет атрибуты идентификации (адресации), которые являются общими для всех средств пакета IPC, а также индивидуальные права доступа, которые обеспечиваются каждым процессом, подключаю-щим этот сегмент.

**Общий список системных вызовов**, обеспечивающих работу с разделяемыми сегментами памяти, имеет вид:



**Создание** нового или получение идентификатора уже существующего разделяемого сегмента памяти осуществляется системным вызовом **shmget(...)**

**Уже созданный** сегмент разделяемой памяти пристыковывается к адресному пространству процесса с помощью системного вызова **shmat(...)**

**Отсоединение**, ранее подключенного сегмента разделяемой памяти, осуществляется системным вызовом **shmdt(...)**

**Общее управление** сегментами разделяемой памяти, с использованием структуры типа shmid\_ds, осуществляется посредством системного вызова **shmctl(...)**

cmd - определяет управляющие команды:

• IPC\_STAT - получить информацию о состоянии разделяемого сегмента, которая копируется в буфер buf;

• IPC\_SET - переустановить характеристики разделяемого сегмента памяти по информации содержащейся в буфере buf;

• IPC\_RMID - удалить разделяемый сегмент, причем пользователь должен быть владельцем, создателем или суперпользователем процесса;

### **57. Передача сообщений.**

**Третьим** и последним средством, входящим в System V IPC, являются очереди сообщений.

**Очереди сообщений** - это наиболее семантически нагруженный способ взаимодействия процессов через каналы связи, в котором на передаваемую информацию накладывается определенная структура, так что процесс, принимающий данные, может четко определить, где заканчивается одна порция информации и начинается другая. Такая модель позволяет задействовать один и тот же канал связи для передачи данных в двух направлениях между несколькими процессами.

**Очереди сообщений**, как семафоры и разделяемая память, являются средством связи с непрямой адресацией, что требует:

• инициализации их, для организации взаимодействия процессов;

• специальных действий, для освобождения системных ресурсов по окончании взаимодействия.

**Пространством имен очередей сообщений** является то же самое множество значений ключа, генерируемых с помощью функции ftok(), а для передачи данных используются **системные примитивы**, в виде функций send() и receive(), которым в качестве параметра передаются IPC-дескрипторы очередей сообщений, однозначно идентифицирующие эти данные во всей вычислительной системе.

**Очереди сообщений** имеют следующие особенности:

• располагаются в адресном пространстве ядра операционной системы в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди.

• каждый элемент списка представляет собой отдельное сообщение.

• каждое сообщения имеет атрибут, называемый типом сообщения.

**Выборка сообщений** из очереди, соответствующая примитиву receive(), может быть выполнена тремя способами:

• В порядке FIFO, независимо от типа сообщения.

• В порядке FIFO, для сообщений конкретного типа.

• Первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

**Реализация примитивов** send() и receive() обеспечивает скрытое от пользователя взаимоисключение, во время помещения сообщения в очередь или его получения из очереди. Она также обеспечивает:

• блокировку процесса, при попытке выполнить примитив receive() над пустой очередью или очередью, в которой отсутствуют сообщения запрошенного типа;

• блокировку процесса, при попытке выполнить примитив send() для очереди, в которой нет свободного места.

Системный вызов **msgget()** предназначен для выполнения операции доступа к очереди сообщений

Системный вызов **msgsnd()** предназначен для помещения сообщения в очередь сообщений.

Системный вызов **msgrcv()** предназначен для получения сообщения из очереди сообщений.

Системный вызов **msgctl()** предназначен для получения информации об очереди сообщений, изменения ее атрибутов и удаления из системы.

### **58. Главный родительский процесс init.**

**Главный родительский процесс** — процесс init:

• первый созданный ядром ОС и последний в работе системы;

• являющийся родителем для всех остальных процессов.

**Уникальные свойства** процесса init делают его особенным в системе:

• он не имеет родительского процесса, а создается ядром ОС;

• обладает свойствами родительского процесса, но не обладает свойствами дочернего процесса;

• выполняет сугубо системные функции, организуя работу других процессов;

• его удаление означает остановку работы ОС.

**Классический подход создания процесса init** предполагает следующую последовательность действий ПО ядра ОС и вспомогательного ПО ЭВМ:

• загрузка ядра ОС с помощью вспомогательного ПО ЭВМ (загрузчика);

• передача управления (функционирования) от загрузчика ядру ОС;

• работа ядра в активном режиме, в котором оно инициализирует свои параметры, создает нужные структуры, инициализирует основные устройства ЭВМ, находит и монтирует корневую файловую систему, создает среду для пользовательского режима, находит в корневой ФС исполняемый файл init (обычно /sbin/init) и, на основе его, создает структуры для первого процесса;

• переключение ядра в пассивный режим осуществляется передачей управления (функционирования) подсистеме управления процессами;

• работа ядра в пассивном режиме, в котором оно, взаимодействуя с оборудованием ЭВМ и переключая процесс, ожидает от процессов запросы на системные вызовы и выполняет их.

**Современный подход создания процесса init** сохраняет преемственность классическому подходу, модифицируя его следующим образом:

• загрузка ядра ОС также осуществляется с помощью загрузчика;

• передача управления (функционирования) от загрузчика ядру ОС дополняется передачей ядру массива данных, содержащих временную ФС;

• работа ядра в активном режиме, в котором оно инициализирует свои параметры, создает нужные структуры, инициализирует основные устройства ЭВМ, распаковывает временную файловую систему и выбирает вариант создания первого процесса: если в корне ФС имеется сценарий init, то создаются структуры для файла /bin/sh, иначе создается среда для файла /sbin/init;

• переключение ядра в пассивный режим осуществляется по классической схеме;

• работа ядра в пассивном режиме осуществляется по классической схеме.

### **59. Четыре подхода к управлению процессами: монопольный, System V, upstart и systemd.**

**Практика использования ОС** потребовала унификации подходов к общей организации вычислительного процесса компьютера и, в первую очередь, разделение работы системного и прикладного ПО.

**Монопольный подход** к управлению процессами применяется:

• во встроенных системах, когда вычислительный процесс решает одну или несколько простых задач, не требующих сложных организационных мероприятий по управлению процессами;

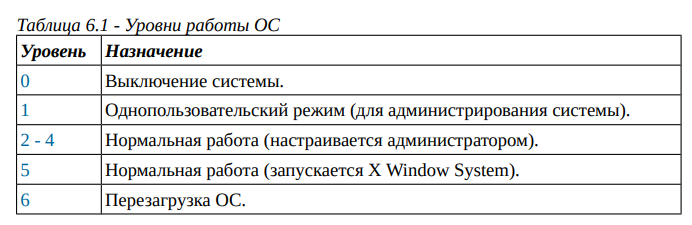
• в узко специализированных системах, требующих централизованного управления всеми вычислительными аспектами приложения;

• в монопольном режиме работы администратора ОС, необходимом для настройки или восстановления работы ОС;

• при загрузке современных ОС, предполагающей промежуточный этап работы ОС перед загрузкой и монтированием основной файловой системы.

**Подход UNIX System V** ориентирован на универсальное применение. Фактически он является классическим примером, организующим управление процессами в UNIX и Linux системах, применяемым и до настоящего времени.

**Основная идея System V** — ядро и ОС в целом могут работать на различных уровнях, показанных в таблице 6.1.



Для определения, на каком уровне работает ОС, используется утилита runlevel, которая выдает два символа, разделенных пробелом:

• первый символ — уровень, на котором система находилась (значение N показывает, что предыдущий уровень не был установлен);

• второй символ - указывает уровень, на котором система находится сейчас.

**Существенными недостатками подхода System V** являются:

• **строго последовательная организация** процедур останова и запуска процессов, которая приводит к существенным задержкам процедур старта и переключения уровней ОС;

• **слабые возможности по отслеживанию** групп процессов, решающих общую задачу, что приводит к потере контроля над отдельными процессами и их незаметное для системы завершение.

**Система инициализации**, основанная на обработке сигналов была реализована в 2009 году Canonical Ltd. для ОС Ubuntu и стала называться **upstart**.

Upstart использует:

• файлы конфигурации, имеющие расширения \*.conf и помещенные в директорию /etc/init/;

• специальное приложение init, помещенное в директорию /sbin/;

• старые скрипты инициализации, находящиеся, для совместимости со старыми версиями, в директории /etc/init.d/.

**Основные понятия upstart**:

• job — работа — общее название запускаемого ПО;

• task - задача - разновидность работы, предполагающая запуск и завершение;

• service - сервис - разновидность работы, аналог демона, которая перезапускается при падении или аварийном завершении.

**Systemd** - демон инициализации, призванный унифицировать управление устройствами и процессами, заменив существующие программные средства, включая init.

**Systemd** является свободным ПО с лицензией GNU v.2.1, который опирается на концепцию сервиса, выделяя:

• сокет-активные и шино-активные сервисы, что часто приводит к лучшему распараллеливанию взаимозависимых сервисов;

• сервисные процессы cgroups, использующие специальные идентификаторы групп, вместо идентификаторов процессов PID, что гарантирует отслеживание главных демонов приложений и не допускает потери процессов, при их разветвлении.

### **60. Порождение и завершение процессов, просмотр состояния и изменение приоритета.**

**Прежде всего,** взаимодействие процессов на этом уровне определяется функциями:

• fork(...) - порождает новый дочерний процесс;

• wait(...) - ожидает завершение любого дочернего процесса, регистрируя их в системе и завершая сам процесс удаления из системы;

• waitpid(...) - ожидает завершение процесса с конкретным номером PID.

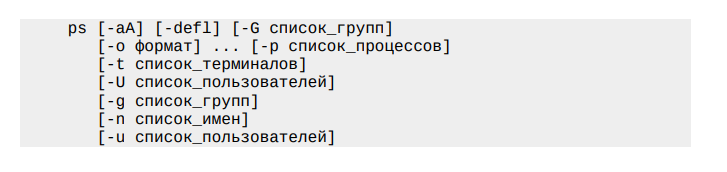
**В случае успешного завершения функция fork(...)** возвращает порожденному процессу 0, а родительскому процессу – идентификатор порожденного процесса.

**Функция wait(...)** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

**Системный вызов \_exit(…)** - обеспечивает завершение процесса, вызвавшего его, удаляет PID процесса из таблицы (списка) процессов и передает родительскому процессу значение статуса завершения.

Для просмотра номеров PID и состояний процессов используются ряд утилит.

**ps (Process status)** — состояние процессов. Выводит на консоль список процессов, включая PID. Формат вывода зависит от используемых ключей.



**top -** Интерактивная программа, соответствующая вызову ps -aux, отображающая процессы в порядке уменьшения потребляемого ими процессорного времени. Перед списком процессов выводится статистика различных характеристик задач и потребляемых ими ресурсов ЭВМ.

**Обычный процесс** при запуске получает приоритет 0. Чтобы запустить процесс с измененной тактичностью, используется команда:

**nice** [ -приращение ] команда аргументы

**Для изменения приоритетов** запущенных процессов, используется команда

**renice** приращение [[ -p ] PID... ] [[ -g ] PGRP...] [[ -u ] username]

**Для управления процессами** также используются команды: прерывания — нажатие клавиш Ctrl-Z;

**bg** — перевод процесса в фоновый режим;

**fg** — перевод процесса в приоритетный режим

### **61. Состояния процессов в ядре ОС.**

**Все процессы ОС** выполняются в режиме пользователя.

Управление работой процессов осуществляет ядро ОС. Для организации управления процессами ядро ОС:

• присваивает процессу уникальный идентификатор PID;

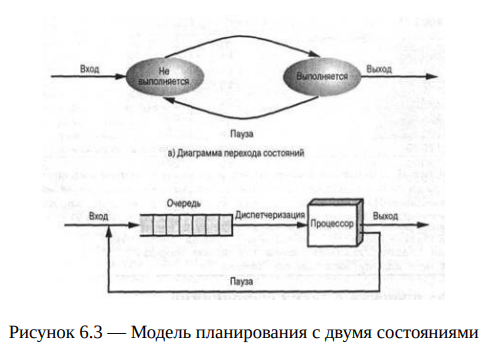
• выделяет процессам для работы некоторые кванты времени t;

• переводит процесс в одно из состояний, обеспечивая мультипрограммный режим работы ОС.

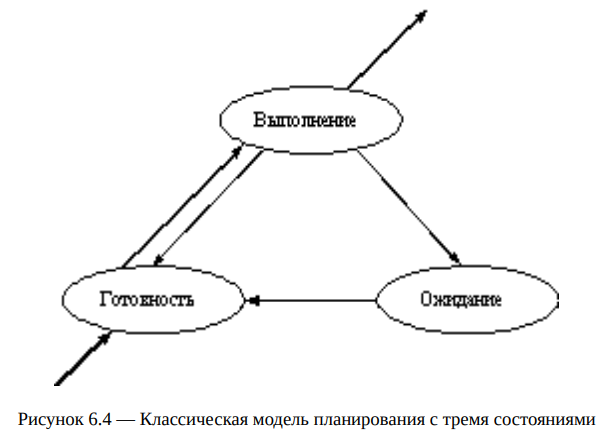
**Для реализации мультипрограммного режима ОС** в подсистеме управления процессами выделена отдельная подсистема - планировщик.

**Планировщик** - часть ПО ядра ОС, который организует некоторую очередь процессов и переключает их в различные состояния, обеспечивая мультипрограммный режим их функционирования.

**Различные реализации планировщиков** отличаются прежде всего количеством состояний процесса, которые он учитывает.



**Классическая для UNIX-систем схема планирования**, показанная на рисунке 6.4, учитывает три состояния: **готовность, выполнение и ожидание.** Она обеспечивает более адекватные алгоритмы планирования.



**Выполнение** - активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором.

**Ожидание** - пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам, он ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса.

**Готовность** - также пассивное состояние процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он готов выполняться, однако процессор занят выполнением другого процесса.

**Более развитая модель планирования**, показанная на рисунке 6.5, содержит пять состояний процесса. Она содержит три состояния классической модели, но еще учитывает создание и завершение процесса, что соответствует полному «жизненному циклу процесса».



### **62. ОС реального времени.**

**Специализация ОС** на решение определенного класса прикладных задач отражается не только в их названии, а в первую очередь влияет на стратегии диспетчеризации.

**Системы пакетной обработки данных** — подход к диспетчеризации, который использовался в первых ОС и супервизорных системах, когда пользователь сам не запускал программу (задачу), а относил ее на ВЦ (вычислительный центр) и забирал результат через некоторый промежуток времени.

**Общая стратегия пакетной обработки данных** - максимально эффективное использование процессорного времени ЭВМ и адекватно представляются моделью с двумя состояниями.

**Часто**, планирование осуществлялось ручным способом:

• маленькие по времени задачи решались в дневное время;

• большие по времени задачи решались ночью.

**Интерактивные системы** - предполагают непосредственное взаимодействие пользователя и ЭВМ, причем обеспечение для пользователя приемлемого темпа диалога, является основным требованием к таким системам. Для этой цели разрабатывается специальное системное и прикладное ПО ОС, которое ориентировано на обеспечение интерактивной работы пользователя. В частности, большое значение имеет развитие графических подсистем ОС, не смотря на то, что они расходуют значительную часть ресурсов ЭВМ.

**Системы реального времени** - допускают интерактивное взаимодействие с пользователем, но ориентированы на автоматическое управление различными техническими системами. Поскольку требования к стратегии планирования определяется требованиями к прикладной задаче управления, то критерии ОС реального времени должны выражаться через терминологию задач.

**Основные требования** к задачам реального времени:

• окончание работы к сроку и исключение потери данных;

• предсказуемость, которая предполагает предотвращение деградации качества в мультимедийных системах.

Общая стратегия планирования в таких системах - запуск на выполнение каждого процесса через интервал времени, не превышающий некоторого значения T.

### **63. Алгоритм разделения времени.**

**Все стратегии планирования** реализуются конкретными алгоритмами планирования.

**В общем случае**, планирование процессов включает решение следующих задач:

• определение момента времени для смены выполняемого процесса;

• выбор процесса на выполнение из очереди готовых процессов;

• переключение контекстов "старого" и "нового" процессов

**Рассмотрим подробнее** две группы наиболее часто встречающихся алгоритмов:

• алгоритмы, основанные на квантовании;

• алгоритмы, основанные на приоритетах.

**В алгоритмах, основанных на квантовании**, смена активного процесса происходит, если:

• процесс завершился и покинул систему;

• произошла ошибка;

• процесс перешел в состояние ОЖИДАНИЕ;

• исчерпан квант процессорного времени, отведенный данному процессу

**Процесс, который исчерпал свой квант,** переводится в состояние ГОТОВНОСТЬ и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых.

**Кванты времени**, выделяемые процессам, могут быть:

• одинаковыми для всех процессов или различными;

• фиксированной величины для одного процесса или изменяться в разные периоды жизни процесса.

**Другая группа** алгоритмов использует понятие "приоритет" процесса.

**Приоритет** - это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии.

**Приоритет может выражаться** целыми или дробными, положительным или отрицательным значением.

**Чем выше привилегии процесса**, тем меньше времени он будет проводить в очередях.

Приоритет может назначаться:

• директивно администратором системы в зависимости от важности работы или внесенной платы;

• вычисляться самой ОС по определенным правилам;

• оставаться фиксированным на протяжении всей жизни процесса: статические приоритеты;

• изменяться во времени в соответствии с некоторым законом: динамические приоритеты.

**Существует две разновидности** приоритетных алгоритмов:

• алгоритмы, использующие относительные приоритеты;

• алгоритмы, использующие абсолютные приоритеты.

**В обоих случаях**, выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет.

**По разному решается проблема** определения момента смены активного процесса:

• в системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ или же произойдет ошибка, или процесс завершится

• в системах с абсолютными приоритетами выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса; в этом случае, прерванный процесс переходит в состояние ГОТОВНОСТЬ

### **64. Язык С как стандарт взаимодействия с ОС.**

Общеизвестно, что язык С рассматривается в двух контекстах:

• как высокоуровневый инструмент для написания самих ОС;

• как низкоуровневый инструмент для написания прикладных программ, взаимодействующих с ядром ОС.

Такое положение языка С делает его особенным, среди других языков программирования, вкладывающим в него не только функциональные возможности по реализации алгоритмов работы арифметики с целыми и вещественными числами, но и набор средств, которые рассматриваются как функции самой ОС. И хотя синтаксис языка не содержит даже собственных конструкций ввода-вывода, функциональное окружение его, реализованное прежде всего в виде библиотеки libc, является неотъемлемой частью ОС и изменяется после перекомпиляции ее ядра. Как следствие указанных особенностей языка С, его реализация для целей программирования сильно привязана как к аппаратной платформе ЭВМ, так и к архитектуре ПО ОС. Это, в свою очередь, порождает проблемы переносимости уже разработанного ПО на разные архитектуры, а также разработку инструментальных средств для этого языка

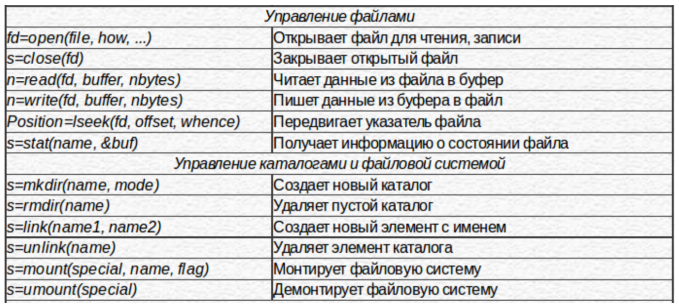
Стремление повысить мобильность ПО, которое пришется на языке С, привело к разработе стандарта **POSIX** (Portable Operating System Interface), призванного стабилизировать описание интерфейсов различных операционных систем. Основные особенности этого стандарта и изложены в данном подразделе.

Стандарт POSIX ориентирован в первую очередь на вызывающую сторону, что делает его полезным как для системных, так и прикладных программистов.

**Цель стандарта POSIX** - сделать приложения мобильными на уровне исходного кода языка. Это значит, что, при переносе программ языка С на другую операционную платформу, потребуется только новая компиляция исходных текстов программ.

Реализация или операционная система, соответствующая стандарту POSIX, должна поддерживать все обязательные служебные программы, функции, а также заголо-вочные файлы с обеспечением специфицированного в стандарте поведения. Для этих целей используется константа ***\_POSIX\_VERSION***, которая имеет значение 200112L.

**65. Системные операции для работы с файловой системой.**



Все системные функции ввода-вывода используют целочисленное значение **дескриптора файла**. Это значение формируется в ядре ОС и индивидуально привязывается к конкретному процессу. Нумерация значений дескрипторов для каждого процесса начинается с нуля и определяется в момент открытия файлов, из условия минимального значения доступного (незанятого) дескриптора. Когда файл закрывается, то соответсвующий ему номер дескриптора освобождается и используется для открытия других файлов.

**66. Создание специальных файлов. Запрос информации о статусе файлов.**

Напомним, что структурную основу EXT2FS, как и других файловых систем Linux/UNIX, играет таблица индексных дескрипторов (Inode Tables), каждая строка которой содержит полное описание файла, кроме его имени:

• когда файл создается в файловой системе, то ему выделяется отдельная строка в Inode Tables;

• номер этой строки — i-узел (inode) и используется для дальнейшей работы с файлом, а имя файла и значение его i-узла записываются в соответствующий файл директории, имеющий записи переменной длинны (см. далее таблицу 1.1);

• когда файл открывается, то VFS находит по имени файла значение i-узла, 30 запоминает его значение и возвращает пользователю номер дескриптора файла;

• пользователь, обращаясь к файлу, указывает его дескриптор файла, а VFS по номеру дескриптора находит значение i-узла файла.

Наряду с изученной ранее функцией open(...), которая позволяет как открывать, так и создавать файлы, в стандарте POSIX имеется функция mknod(...), обеспечивающая возможность создавать и различные специальные файлы.

**Специальные файлы**, после их создания, получают свои номера i-узлов, но, в отличие от обычных файлов, они в действительности есть только указатели на соответствующие драйверы устройств в ядре. По сравнению с обычными файлами файлы устройств имеют три дополнительных атрибута, которые характеризуют устройство, соответствующее данному файлу: класс устройства, старший номер устройства и младший номер устройства.

*Класс устройства* фактически соответствует понятию тип устройства, определенному выше.

*Старший номер устройства*, группирует типы устройств, например, жесткий диск или звуковая плата.

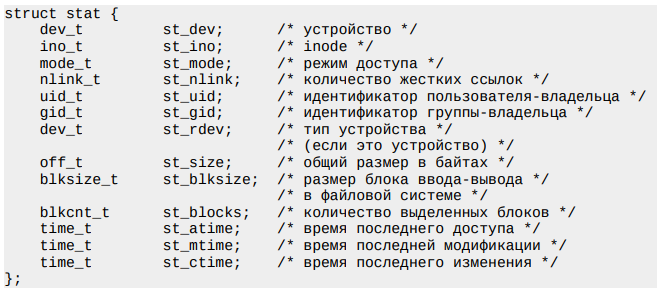
*Младший номер устройства* применяется для нумерации устройств одного типа, другими словами, - устройств с одинаковыми старшими номерами.

Изучив основные особенности файлов, перейдем к рассмотрению системного вызова **stat**(...), который по заданному имени файла позволяет получить информацию о свойствах любых типов файлов.

Общее описание перечисленных функций имеет следующий вид:



Все эти функции возвращают структуру stat, которая содержит следующие поля:



**67. Каналы. Дублирование дескрипторов файлов. Монтирование и демонтирование ФС.**

**Каналы** - старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий операционной системы UNIX. Они предоставляют метод односторонних коммуникаций между процессами: отсюда появился термин half-duplex channel.

Сам **канал создается** в ядре ОС. Когда процесс создает канал, ядро устанавливает **два файловых дескриптора** для пользования этим каналом. Один такой дескриптор используется, чтобы открыть путь ввода в канал (**запись**), в то время как другой применяется для получения данных из канала (**чтение**)

Создание **неименованного канала** выполняется с помощью функции pipe(...).

Основное **назначение неименованных полудуплексных каналов** — взаимодействие между родительским и дочерними процессами. В этом случае используется то свойство, что дочерний процесс наследует все ресурсы, открытые родительским процессом, в том числе и открытые каналы.

Для целей взаимодействия не родственных («чужих») процессов предназначены **именованные каналы FIFO**, которые также создаются в ядре ОС, но имеют имена, отображаемые в файловой системе.

Типичное применение каналов FIFO - разработка приложений «клиент — сервер», когда:

• несколько процессов могут записывать или читать FIFO одновременно;

• режим работы с FIFO - полудуплексный, что позволяет процессам общаться только в одном из направлений.

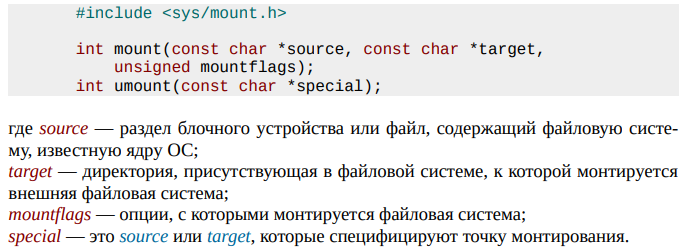
Стандарт POSIX предоставляет для **создания именованных каналов** специальный системный вызов **mkfifo**(...).

В практических задачах часто возникает необходимость **дублирования дескрипторов** или открытие дескриптора с заданным номером. Стандарт POSIX предоставляет два системных вызова дублирования дескрипторов файлов:



где oldfd — старый дескриптор файла; newfd — новый дескриптор файла.

Все ОС имеют системные вызовы, предназначенные для подключения и отключения внешних файловых систем к корневой файловой системе. Стандарт POSIX определяет эти функции как:



При удачном завершении вызова возращаемое значение равно нулю. При ошибке возвращается -1.

**68. Ссылки на имена файлов.**

Для создания новых имен, которые интерпретируются как имена файлов, стандарт POSIX предоставляет два системных вызова:

• **link**() - создает новую ссылку на существующий файл (на i-узел) внутри одной файловой системы, известную также как "жесткая" ссылка; она имеет теже свойства, что и старое имя, поэтому их оригинальность их установить невозможно;

• **symlink**() - создает символьную ссылку на имя предположительно существующего файла, независимо от файловой системы, в которой оно должно находиться; если такого файла не существует, то ссылка называется «висячей».

**69. Классификация способов управления памятью ОС.**

С появлением и совершенствованием первых ОС, которые также, как и прикладное ПО, являются программным обеспечением, стали применять три способа загрузки:



Развите этих способов загрузки ПО показано слева на право:

• певый способ предполагает первоначальную загрузку ОС, обычно в младшие адреса ОЗУ; такой способ применялся в мэйнфреймах и первых миникомпьютерах; программы пользователя загружались с помощью ПО ОС в остальную часть ОЗУ;

• второй способ стал применяться в некоторых карманных компьютерах и встроенных системах, которые обладают малыми ресурсами и, как правило, разными конструктивными особенностями; здесь ОС пишется в ПЗУ, которое соответствует верхним адресам памяти, оставляя ОЗУ полностью для ПО пользователей;

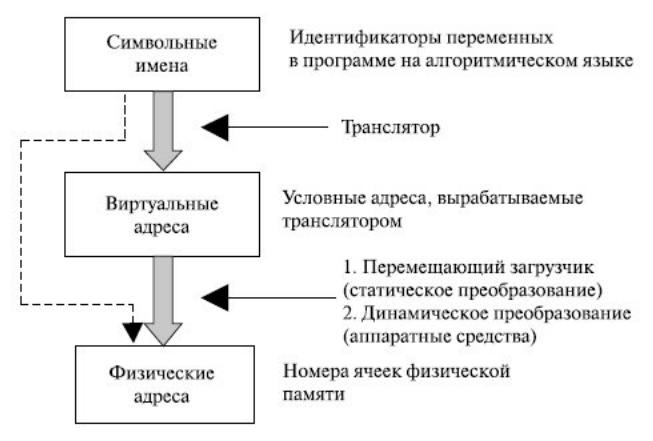
• третий способ соответствует большинству современных систем, подобных IBM PC; в ПЗУ записывается ПО BIOS, которое работает только при включении питания компьютера; ПО BIOS находит и загружает в младшие адреса ОЗУ ПО ОС; программам пользователей предоставляется оставшееся ОЗУ до 640 Кбайт, в младших адресах, и вся оставшаяся память свыше первого Мбайт; память ОЗУ между 640 Кб и 1 Мб (скрытая область) распределяется между ПО BIOS и памятью прямого доступа для внешних устройств.

Для компьютера, память (ОЗУ) всегда была и остается дефицитом:

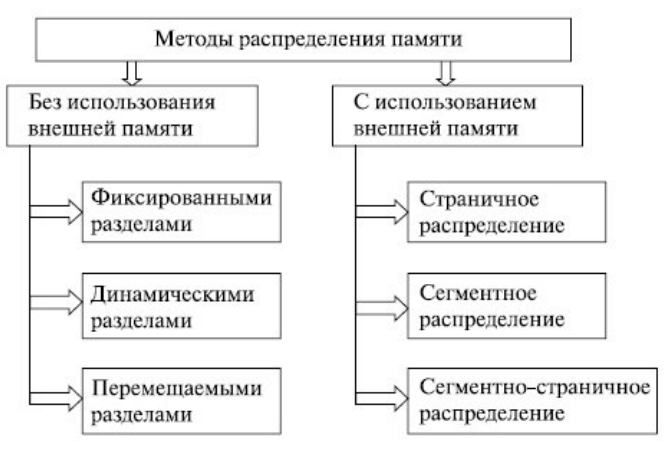
• всегда имеются программы, которым, по тем или иным причинам, недостаточно выделенного объема ОЗУ ЭВМ;

• проблемы реализации языков программирования требуют решать многие вопросы, связанные с адресацией памяти и ее распределением.

Для решения этих и многих других вопросов было введено понятие виртуальной памяти. Виртуальная память, идея которой наглядно показана на рисунке ниже, призвана разрешить противоречие в адресации команд и данных, связанных с потребностями прикладного ПО и ресурсами ОЗУ, которые имеет конкретная ЭВМ.



Появление ОС, поддерживающих мультипрограммирование, потребовало решения вопросов распределения памяти ОЗУ между многими одновременно выполняемыми программами (процессами). На рисунке ниже приведена классификация различных подходов, решающих поставленную проблему



**Методы, не использующие внешнюю память**, предполагают, что ОЗУ достаточно для решения самой крупной задачи, или, что программу можно исполнять по частям, для которых ОЗУ достаточно. Здесь, в большинстве случаев, задачи распределения памяти ставятся как оптимальное разбиение ОЗУ на сегменты (отдельные и не пересекающиеся участки памяти), обеспечивающие максимальную эффективность использования процессорного времени ЭВМ.

**Методы, использующие внешнюю память**, предполагают, что ОЗУ расширяется за счет памяти медленных внешних устройств и становится возможным одновременное выполнение необходимого числа «параллельно» работающих процессов.

Здесь, в большинстве случаев, задачи распределения памяти ставятся как ограничения на скорость выполнения отдельных процессов или эффективность использования ОЗУ ЭВМ по управлению самим процессом распределения памяти ЭВМ

**70. Программный и аппаратный способы управления памятью.**

**Программный способ адресации** неявно присутствует в любом исходном тексте программы в виде операторов команд программы, которые обращаются к объявленным переменным данных и осуществляют различные управляющие действия по определению порядка выполнения самих команд. В последствии, компилятор обеспечит явное присутствие как адресов команд, так и адресов данных. Соответственно, в каждом отдельном процессе, минимально можно выделить:

• управляющий блок процесса;

• сегмент кода (программы);

• сегмент данных;

• стек;

• «куча», если процесс использует динамическое выделение памяти.

Аппаратный способ адресации заложен сам процессор:

• специализацией регистров процессора: сегментные, смещения и индексные;

• методами адресации команд процессора;

• шинами (каналами) процессора.

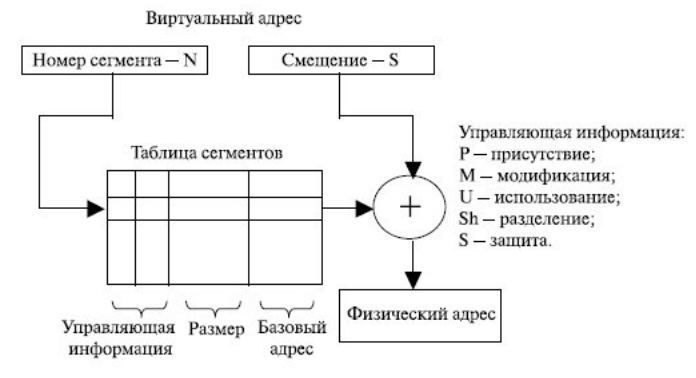
**71. Страничная и сегментная адресации памяти.**

**Сегментная адресация ОЗУ** реализуется с помощью специальных регистров процессора, которые так и называются: регистры сегментов и регистры смещения.

Сегментный способ адресации ОЗУ всегда использует таблицу сегментов, которая сама размещается в ОЗУ и предназначена для решения двух основных задач:

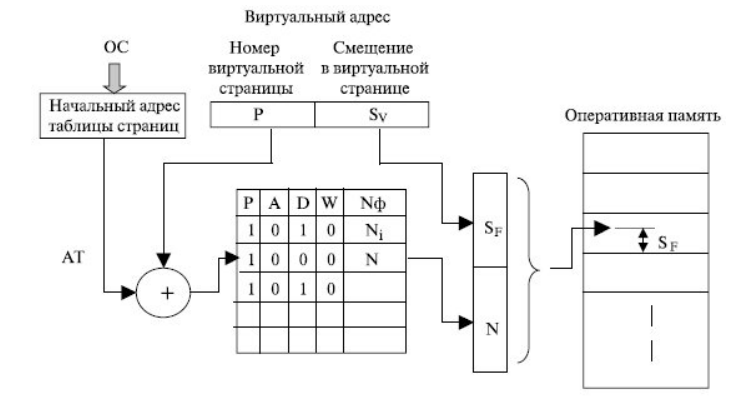
• виртуализация относительных адресов программы относительно физической памяти ЭВМ;

• управление свопингом и перемещение сегментов в физическом пространстве ОЗУ.



Идея использования блоков одинакового размера, для адресации физической и логической частей ОЗУ, стала называться **страничной организацией памяти**. Основное достоинство страничной адресации ОЗУ - отсутствие фрагментации, что не требует затрат ресурса ЭВМ на ее устранение.

Как показано на рисунке ниже, схема адресации ОЗУ с помощью страниц очень похожа на схему адресации ОЗУ с помощью сегментов, поскольку в обоих случаях используются таблицы переадресации, которые должны размещаться в той же ОЗУ.



Основная проблема адресации с помощью страниц - большой размер таблиц адресации, которые должны храниться в ОЗУ. Другая проблема, с которой сталкиется страничный способ адресации — значительные затраты времени, связанные с поиском информации о странице в больших таблицах страниц.

**72. Комбинированный способ адресации памяти.**

Преимущества адресации имеются у каждого способа:

• сегментная адресация памяти — требует относительно малое число записей в таблице сегментов.

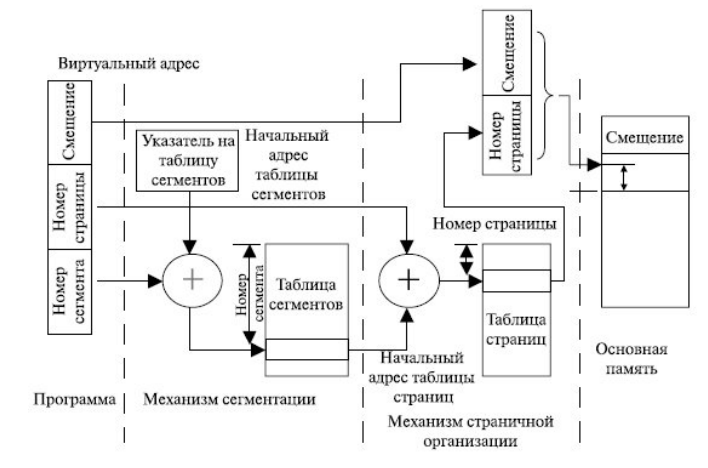
• страничная адресация памяти — устраняет проблемы фрагментации, поскольку ОЗУ адресуется с помощью произвольного доступа, следовательно, свободную страницу ОЗУ можно найти в любом месте памяти.

Отсюда возникает идея смешанной (комбинированной) адресации, которая состоит в том, что:

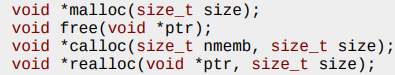
• отдельная программа (процесс) отображается в виде нескольких записей в общей таблице сегментов;

• кажный сегмент, выделенный программе, отображается внутри нее в виде 19 одной или нескольких таблиц страниц.

Общая схема такой адресации приведена на рисунке ниже

**73. Системные вызовы ОС по управлению памятью.**

В данном подразделе, мы изучим системные вызовы ОС, обеспечивающие выделение памяти для нужд программиста. Для этих целей существует ряд функций, позволяющих выделять и освобождать память ОЗУ динамически, во время выполнения программы. Эти функции имеют следующее определение:



Семантика этих функций - следующая:

• malloc(...) - выделяет область неинициализированной памяти, размер которой указан в байтах, в качестве аргумента, и возвращает значение указателя на эту область; если значение аргумента равно нулю, то возвращается значение указателя NULL;

• free(...) — освобождает память ОЗУ, которая ранее была выделена одной из функций: malloc(...), calloc(...) или realoc(...); если аргумент функции равен NULL, то никаких действий не производится;

• calloc(...) - выделяет память ОЗУ под массив данных, размерность которого задана первым аргументом, а размер каждого элемента массива в байтах, задан вторым аргументом; при этом, значение каждого байта выделенной памяти обнуляется; возвращает указатель на выделенный массив данных; если значение любого из аргументов функции равно нулю, то память ОЗУ процессу не выделяется и возвращается значение NULL;

• realloc(...) - изменяет размер уже выделенного блока памяти, используя первый аргумент как указатель, а второй — как нужный размер; если размер области уменьшается, то значения байт оставшейся области не изменяется; если размер области увеличивается, то добавляемая область не инициализируетсся; если значение аргумента указателя равно NULL, то результат функции, соответствует действию функции malloc(...); если требование на новый размер памяти равно нулю, а указатель не равен NULL, то действие эквивалентно функции free(...).

### **74. Подсистема управления процессами.**

Подсистема управления процессами находится в ядре ОС. Основная ее функция — обеспечение мультипрограммного режима работы ОС, что связано с:

• созданием процессов в системе и удаление их из системы;

• переключением процессов в режимы «Готовность», «Выполнение» и «Ожидание»

Качественное выполнение этой функции требует планирования подсистемой своих действий, что в общем случае не является однозначно решаемой задачей.

Чтобы оценить сложность решаемой задачи планирования, рассмотрим перечень требований предъявляемых к ней, в зависимости от целевых аспектов различных прикладных систем.

Все системы должны обеспечить:

• Справедливость — предоставление каждому процессу справедливой доли процессорного времени.

• Принудительное применение политики - контроль за выполнением принятой политики;

• Баланс — поддержка занятости всей системы.

Системам пакетной обработки данных необходима:

• Пропускная способность — максимальное количество задач в час.

• Оборотное время — минимизация времени, затрачиваемого на ожидание обслуживания и обработку задачи.

• Использование процессора — поддержка постоянной занятости процессора.

Интерактивным системам важно:

• Время отклика — быстрая реакция на запросы.

• Соразмерность — выполнение пожеланий пользователя.

Системам реального времени требуется:

• Окончание работы к сроку — предотвращение потери данных.

• Предстказуемость - предотвращение деградации качества в мультимедийных системах.

Планирование в системах пакетной обработки данных:

• Первым пришел — первым обслужен. Является наиболее простым алгоримом планирования, который выделяет первому процессу, запросившему процессор, все время, необходимое для его завершения.

• Кратчайшая задача — первая. Позволяет очень быстро выполнять маленькие задачи, но требует знания времени их выполнения.

• Наименьшее оставшееся время выполнения. Если имеется задача, время выполнения которой меньше, чем время завершения текущей, то текущая задача останавливается, а минимальная, по времени исполнения, запускается. Здесь требуется также знать время выпонения процессов.

• Трехуровневое планирование. Здесь имеется впускной планировщик, который выбирает задачи из общей очереди и передает их процессору на выполнение. Возможны разные варианты, учитывающие возможности процессора и устройств ввода-вывода. Второй уровень планирования определяет: какие процессы можно хранить в памяти, а какие — на диске. Этим занимается планировщик памяти.

Планирование в системах интерактивной обработки данных также обладает большим разнообразием. Наиболее известных два алгоритма:

• Циклическое планирование, когда каждому запускаемому процессу выделяется квант времени, по истечении которого или по запросу устройств вводавывода процесс останавливается и помещается в конец очереди.

• Приоритетное планирование, когда каждому запускаемому процессу присваивается приоритет и управление передается готовому к работе процессу, с наивысшим приоритетом.

Планирование в системах реального времени, которые подразделяются на:

• Жесткие системы реального времени, требующие жестких сроков реакции на запросы каждой задачи.

• Мягкие системы реального времени, для которых нарушение сроков выполнения задач - нежелательно, но допустимо

**75. Синхронизация процессов.**

Вопросы синхронизации процессов описывают проблематику взаимодействия активных элементов ПО, связанных единым алгоритмом реализации их работы. Основы такой синхронизации заложены в самой модели процесса:

• процесс создается на основе родительского процесса, наследуя от него программный код и все открытые ресурсы;

• родительский процесс отслеживает завершение дочернего процесса, тем самым синхронизируя иерархию процессов и разгружая ядро ОС от прикладных аспектов взаимодействия процессов.

Следующей по важности проблемой является реакция процессов на события, которые по своей природе являются асинхронными (случайными) и не могут быть эффективно реализованы в прикладном алгоритме программы. Более того, реакция на события должна распространяться на все процессы, которые не связаны между собой «родственными» отношениями. «Механизмом» такой синхронизации процессов являются сигналы, перечень которых должен поддерживаться ядром ОС.

Дальнейшее развитие средств синхронизации процессов связано с понятем канала как реализация части функций «Подсистемы ввода-вывода». Очевидно, что реализация этих каналов тесно связана и с «Подсистемой управления процессами», подчеркивая сложность реализации функций ядра ОС.

Недостатки такой синхронизации:

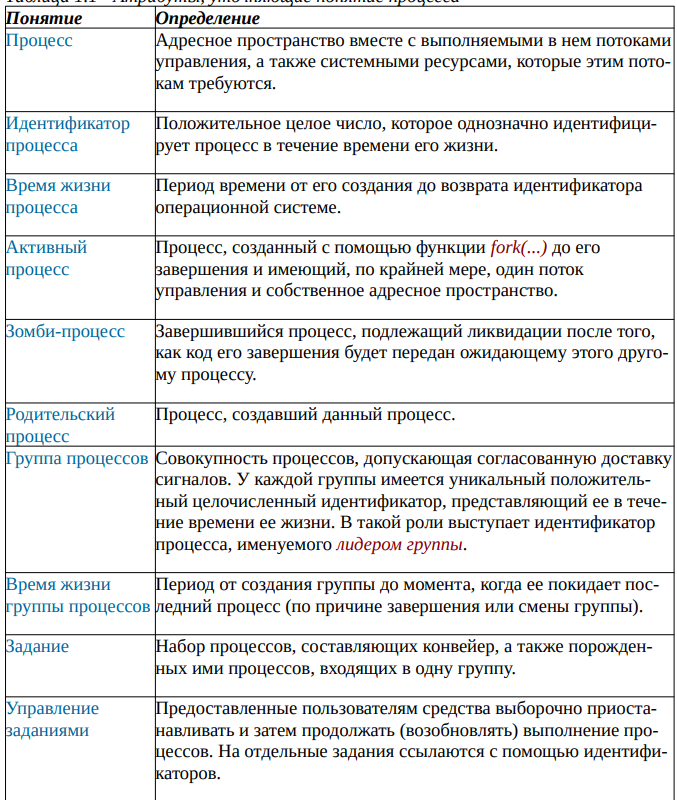
• неформатированный обмен данными, что требует выделения переданного сообщения на каждой взаимодействующей стороне;

• необходима дополнительная синхронизация, даже при реализации схемы один-ко-многим.

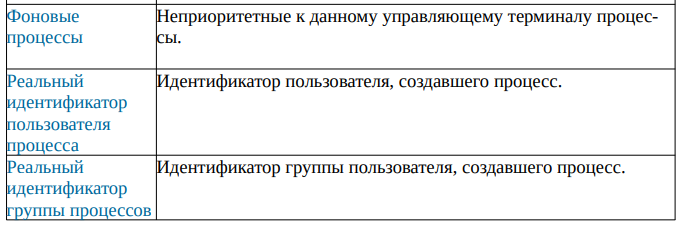
Принципиальное для многих задач решение синхронизации связано с моделью потоков (нитей, threads), которые обеспечивают прикладной программе общее адресное пространство как для кода, так и для данных. Сама синхронизация возлагается на алгоритм программы, но, в отличие от неименованных каналов, не требует организации средств передачи сообщений, а также специального форматирования передаваемых данных. Существенный недостаток использования каналов - невозможность взаимодействия произвольных процессов ОС.

Общим недостатком всех базовых средств взаимодействия процессов является отсутствие полных гарантий на заданную последовательность выполняемых операций в системе. Причина состоит в том, что все указанные средства синхронизации реализуются ядром ОС на фоне действий планировщика процессов. Как следствие, невозможно определить какой из процессов первым захватит нужный многим процессам ресурс или изменит какие-либо данные

### **76. Стандарты POSIX.**







**77. Системные вызовы ОС по управлению процессами.**



**78. Системный вызов fork() и каналы процессов.**

Новый (порожденный, дочерний) процесс является точной копией (родительского) процесса, вызвавшего fork(...), за исключением следующих моментов:

• у порожденного процесса свой идентификатор, отличный от идентификатора родительского процесса;

• у порожденного процесса собственная копия файловых дескрипторов, ссылающихся на те же описания открытых файлов, что и соответствующие дескрипторы родительского процесса;

• порожденный процесс не наследует блокировки файлов, установленные родительским процессом.

• порожденный процесс создается с одним потоком управления – копией того, что вызвал fork(...);

В случае успешного завершения функция fork(...) возвращает порожденному процессу 0, а родительскому процессу – идентификатор порожденного процесса. После этого оба процесса начинают независимо выполнять инструкции, расположенные за обращением к функции fork(...).

### **79. Нити (threds).**

Взаимодействие процессов посредством каналов, является мощным инструментальным средством ОС, но имеет ряд существенных недостатков:

• необходимость синхронизации операций чтения и записи в каналы для разных асинхнонно выполняющихся процессов;

• необходимость форматирования передаваемых в канал данных;

• необходимость дешифровки читаемых из канала данных.

Во многих современных ОС существует расширенная реализация понятия процесс, когда исполняемый образ программы представляет собой не только совокупность выделенных ему ресурсов данных и каналов взаимодействия, но и **набор нитей исполнения**.

Каждый, запущенный ОС, процесс имеет одну нить исполнения, которая называется главной или начальной нитью. Дополнительные нити (threads) создаются программистом из главной нити посредством набора системных вызовов.

Все дополнительные нити процесса разделяют:

• его программный код;

• глобальные переменные;

• системные ресурсы.

Но каждая дополнительная нить имеет:

• собственный программный счетчик;

• свое содержимое регистров;

• свой стек

Для работы с нитями разработано большое количесво функций. Далее, мы рассмотрим только функции, которые реализуются следующими системными вызовами ОС:

• pthread\_create() - создание нити;

• pthread\_exit() - завершение работы нити;

• pthread\_join() - другой вариант завершения работы нити;

• pthread\_self() - получение идентификатора нити.

**80. Сигналы POSIX.**

Формально, стандарт POSIX-2001 под **сигналом** понимает «механизм», с помощью которого процесс или поток управления уведомляют о некотором событии, произошедшем в системе, или подвергают воздействию этого события.

Говорят, что сигнал генерируется (или посылается) для процесса (потока управления), когда происходит вызвавшее его событие:

• выявлен аппаратный сбой;

• отработал таймер;

• пользователь ввел с терминала специфическую последовательность символов;

• процесс обратился к функции kill(...) и другие.

Говорят, что сигнал доставлен процессу, когда взято для выполнения действие, соответствующее данным процессу и сигналу. В интервале, от генерации до доставки или принятия сигнал называется ждущим

В данном подразделе мы завершаем рассмотрение базовых системных вызовов управления процессами, объединенными в группу «Функции потоков и сигналов процессов», где будут изучены следующие системные вызовы:

• signal() - установка нового обработчика сигнала текущего процесса;

• kill() - посылка сигнала процессу по его иденитфикатору PID;

• raise() - посылка сигнала текущему процессу;

• pause() - остановка процесса до получения сигнала;

• sleep() - остановка процесса на заданное число секунд, либо до получения сигнала.

### 

### **81. Проблемы распределения ресурсов ОС.**

Задачи взаимодействия процессов можно свести к ***задачам борьбы за ресурсы ЭВМ***. Таким ресурсом может быть память ЭВМ или очередь процессов на печать. В борьбе за общий разделяемый ресурс, процессы входят в ***состояние состязания***.

***Состояние состязания*** - ситуация, в которой два или более процессов считывают или записывают данные одновременно, а результат зависит от того, кто из них был первым.

Основной способ решения этих проблем - ***взаимное исключение***.

***Взаимное исключение*** предполагает, что одновременно не могут выполняться две ***критические секции*** разных процессов.

***Критической секцией*** называется часть кода процесса, который обеспечивает доступ к разделяемым ресурсам.

Чтобы сформулировать общую проблематику использования любых разделяемых ресурсов, были сформулированы четыре необходимых ***условия***:

* два процесса не должны находиться в критических областях;
* в программе не должно быть предположений о скорости или количестве процессов;
* процесс, находящийся в критической области, не может блокировать другие процессы;
* невозможна ситуация, когда процесс вечно ждет попадания в критическую область.

Среди известных способов, реализующих взаимное исключение, можно выделить следующие:

* ***запрет всех прерываний***, когда процесс вошел в критическую область;
* ***переменные блокировки***, когда процесс, прежде чем войти в критическую область, использует значение переменной, которая доступна всем процессам. Если значение переменной равно 0, то процесс изменяет ее значение на 1 и входит в критическую область, а если значение переменной равно 1, то процесс ждет, пока ее значение не изменится на 0;
* ***строгое чередование***, когда процесс имеющий и использующий критическую область становится в очередь и получает номер этой очереди, в цикле проверяя некоторую общую переменную, значение которой указывает номер очереди с разрешенным входом в критическую область. Когда процесс выходит из критической области он изменяет значение этой переменной, тем самым, разрешая другому процессу войти в критическую область.
* ***алгоритм Петерсона***, который использует две функции enter\_region(int process) и leave\_region(int process). Вызов первой является обязательным условием, при входе процесса в критическую область. Она также сохраняет дополнительные данные о процессах, вызвавших ее, поэтому только один процесс может войти в критическую область, а другому необходимо ждать, когда выходящий из критической области процесс вызовет функцию leave\_region(int process), освобождающую доступ к нужной области.
* ***команда TSL*** (Test and Set Lock — проверить и заблокировать), которая на аппаратном уровне гарантирует неделимость операций чтения и записи слова памяти, обеспечивая однозначность отражения блокировки процессом разделяемого ресурса.

***Пример*** простейшей проблемы разделения общих ресурсов:

***Проблема производителя и потребителя***, которая соответствует взаимодействию двух процессов, использующих ***общий буфер данных*** (***проблема ограниченного буфера***).

Более конкретно:

* один из процессов, **производитель**, помещает данные в буфер;
* другой процесс, **потребитель**, читает данные из буфера;
* трудности начинаются, когда производитель хочет поместить данные в буфер, а он уже заполнился; тогда производитель должен ждать, когда второй процесс, полностью или частично, не освободит буфер;
* аналогично, когда потребитель хочет прочитать данные из буфера, а он пуст, процесс уходит в ожидание, пока другой процесс не положит данные в буфер и не разбудит потребителя.

Ситуация значительно осложняется, когда процессу необходимо захватить несколько разделяемых ресурсов. В этом случае, возникает ***состояние взаимной блокировки***.

***Состояние взаимной блокировки*** возникает тогда, когда процесс успел захватить только часть разделяемых ресурсов, необходимых ему для выполнения работы, а другую часть захватили другие процессы. В такой ситуации, все процессы переходят в состояние бесконечного ожидания необходимых им ресурсов.

**Примером** задачи, приводящей к взаимным блокировкам, является «**Задача об обедающих философах**».

Одним из первых системных средств, обеспечивающих программистов надежными инструментами работы с взаимодействующими процессами, стал пакет, имеющий название ***System V IPC***. В последующем, его идеи были положены в средства сетевого взаимодействия ЭВМ.

### **82. Системный пакет IPC.**

С целью обеспечить программистов надежным инструментом, устранения проблем взаимодействующих процессов, было разработано ряд системных средств ядра ОС, известных под общим названием ***System V IPC***.

***IPC*** (Inter Process Communication) — ориентирован на решение трех проблем:

* надежная и простая передача данных и сообщений от одного процесса другому;
* контроль над деятельностью процессов в критических ситуациях;
* согласование действий между процессами. Непосредственная реализация пакета разделена на четыре части:
* средства адресации IPC;
* семафоры;
* разделяемая память и очереди сообщений.

В пакете ***IPC*** предусмотрена система асинхронного взаимодействия процессов, посредством системы адресации, которой должен пользоваться каждый процесс, использующий средства пакета ***IPC***. Необходимость такой адресации обусловлена тем, что процессы должны различать разные виды взаимодействия.

В качестве информационной основы формирования таких адресов используются:

* ***имена файлов***, доступные в файловой системе ОС;
* ***произвольное целое число***, отличное от нуля, интерпретируемое как номер проекта.

Системный вызов, создающий адрес (**идентификатор, ключ**) для пакета **IPC**, имеет вид:



***pathname*** - должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию;

***proj\_id*** – это целое число, характеризующее экземпляр средства связи.

В случае удачного завершения вызова возвращается значение созданного ключа типа ***key\_t***. При ошибке возвращается ***-1***, а в переменную ***errno*** записывается код ошибки, согласно системному вызову ***stat(2)***.

Ядро ОС хранит информацию обо всех средствах ***System V IPC***, используемых в системе. В прикладном плане, ключ, генерируемый функцией ***ftok(...)***, рассматривается как средства связи между взаимодействующими процессами.

При создании нового средства связи или получении доступа к уже существующему, процесс получает неотрицательное целое число - ***дескриптор*** (идентификатор) этого средства связи, которое однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор должен передаваться в качестве параметра всем системным вызовам, осуществляющим дальнейшие операции над соответствующим средством ***System V IPC***.

### **83. Утилиты управления средствами пакета IPC.**

Для работы со средствами межпроцессного взаимодействия System V IPC, имеются три основные утилиты, которыми следует пользоваться по мере необходимости:

* **ipcmk** — создание различных ресурсов средств IPC;
* **ipcs** - вывод отчёта о состоянии средств межпроцессного взаимодействия;
* **ipcrm** - удаление очередей сообщений, наборов семафоров и разделяемых сегментов памяти.

***Утилита ipcmk*** позволяет открывать разделяемые сегменты памяти, очереди сообщений и массивы семафоров.

ipcmk [options]

Имеются следующие **опции**:

* -M, --shmem size Открытие разделяемого сегмента памяти размером size байт;
* -Q, --queue Открытие очереди сообщений;
* -S, --semaphore number Открытие массива семафоров с number элементами;
* -p, --mode mode Установка доступа к ресурсу; по-умолчанию = 0644;
* -V, --version Отображение версии пакета;
* -h, --help

***Утилита ipcs*** выводит информацию о системных средствам межпроцессного взаимодействия System V (IPC).

ipcs [-abcmopqstMQSTy] [-C дамп] [-N система] [-u пользователь]

Имеются следующие **опции (**опишу основные, а то их многа):

* *-a* Показать максимально возможное количество информации во время вывода данных об активных семафорах, очередях сообщений и разделяемых сегмен- тах памяти. Это эквивалентно указанию опций -b, -c, -o, -p и -t.
* -m Вывести информацию об активных сегментах разделяемой памяти.
* -q Вывести информацию об активных очередях сообщений.
* -s Вывести информацию об активных семафорах.
* -C дамп Извлечь значения из списка имён (namelist) указанного дампа памяти ядра, вместо определённого по умолчанию /dev/kmem. Подразумевает -y.
* -N система Извлечь список имён из указанной системы, вместо определённой по умолчанию /boot/kernel/kernel. Подразумевает -y.
* -T Вывести системную информацию о разделяемой памяти, очередях сообщений и семафорах.
* -y Использовать интерфейс kvm вместо интерфейса sysctl для извлечения необходимой информации. Если ipcs запущена на работающей системе, использование kvm(3) потребует привилегии чтения из /dev/kmem.
* -u пользователь Вывести информацию о механизмах IPC для указанного пользователя. Пользователь может быть задан либо числовым идентификатором UID, либо регистрационным именем.

Если не указана ни одна из опция -M, -m, -Q, -q, -S, или -s, то выводится информация обо всех активных средствах IPC.

***Утилита ipcrm*** удаляет указанные очереди сообщений, семафоры и разделяемые сегменты памяти из системы. Требуемые объекты System V IPC задаются идентификатором их создания или любым связанным с ними ключом.

ipcrm [-q msqid] [-m shmid] [-s semid] [-Q msgkey] [-M shmkey] [-S semkey] …

Для выбора объектов IPC, которые будут удалены, используются следующие опции, которых может быть задано любое число и любая комбинация:

* -q msqid Удалить из системы очередь сообщений, связанную с идентификатором msqid.
* -m shmid Пометить для удаления разделяемый сегмент памяти, связанный с идентификатором shmid. Этот помеченный сегмент будет уничтожен после отключения от него последнего процесса.
* -s semid Удалить из системы набор семафоров, связанный с идентификатором semid.
* -Q msgkey Удалить из системы очередь сообщений, связанную с ключом msgkey.
* -M shmkey Пометить для удаления разделяемый сегмент памяти, связанный с ключом shmkey. Этот помеченный сегмент будет уничтожен после отключения от него последнего процесса.
* -S semkey Удалить из системы набор семафоров, связанный с ключом semkey

### **84. Семафоры.**

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали ***семафоры***, концепцию которых описал Дейкстра. При разработке средств ***System V IPC семафоры*** вошли в их состав как неотъемлемая часть.

Набор действий над семафорами System V IPC включает три **операции**:

* A(S, n) — семафор S увеличивается на n;
* D(S, n) – пока значение семафора S < n, процесс блокируется. Далее, выполняется S = S - n;
* Z(S) – процесс блокируется до тех пор, пока значение семафора S не станет равным 0.

Основная идея реализации этого механизма, предполагает, что:

* семафор - это **минимальный примитив синхронизации**, служащий основой для более сложных механизмов синхронизации, определенных в прикладной программе;
* у семафора есть **значение**, которое представляется целым числом в диапазоне от 0 до 32767;
* прикладная реализация механизма синхронизации обеспечивается **набором (массивом) семафоров**, операции над этими наборами, для приложений являются атомарными;
* **гарантом атомарности** операций на наборами является ядро ОС, в котором и реализованы механизмы синхронизации.

В пакете **IPC**, работа с семафорами осуществляется с помощью трех системных вызовов:



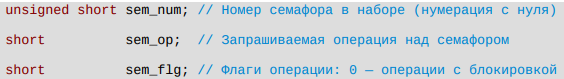
int **semget**(key\_t **key**, int **nsems**, int **semflg**) предназначен для выполнения операции доступа к массиву IPC-семафоров или его создание. В случае его успешного завершения, возвращается дескриптор System V IPC для этого массива. В случае ошибки возвращается -1.

* где **key** — ключ, ассоциированный с семафором и генерируемый функцией **ftok**(...);
* **nsems** - задает число семафоров в наборе;
* **semflg** — флаг семафора; обычно ассоциируется с константами:

int **semop** (int **semid**, struct sembuf \***sops**, size\_t **nsops**) предназначен для выполнения операций **A**, **D** и **Z** на основе ключа, который получен системным вызовом **semget**(...). При успешном завершении возвращает 0, иначе -1.

* где **semid** — идентификатор набора семафоров, созданный функцией semget(...);
* **sops** — указатель на массив структур, с числом элементов **nsops**, состоящим из структур типа **sembuf**;
* **sops** — указатель на массив структур, с числом элементов **nsops**, состоящим из структур типа **sembuf**;

Действия, которые выполняются над набором семафоров, определяются вторым аргументом системного вызова **semop**(...), являющимся указателем на массив структур типа **sembuf**. Каждая структура типа **sembuf** содержит, по крайней мере, следующие три поля:



Операция над конкретным семафором определяется значением поля **sem\_op**:

* положительное значение предписывает увеличить значение семафора на указанную величину, соответствует операции **A(S,n)**;
* отрицательное - уменьшить, соответствует операции **D(S,n)**;
* нулевое - сравнить с нулем, соответствует операции **Z(S)**.

С точки зрения **пользовательского процесса**, выполнение операций над массивом семафоров является неделимым действием. Это значит:

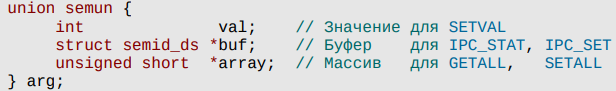
* если операции выполняются, то только все вместе; 16
* никакой другой процесс не может получить доступ к промежуточному состоянию набора семафоров, когда часть операций из массива уже выполнилась, а другая еще не успела.

Сама ОС выполняет операции из массива семафоров - по очереди, причем порядок их обработки - не оговаривается. Если очередная операция не может быть выполнена, то:

* эффект предыдущих операций аннулируется;
* вызов функции semop(...) приостанавливается (операция с блокировкой, когда sem\_flg=0) или немедленно завершается неудачей, когда выполняется операция без блокировки.

int **semctl** (int **semid**, int **semnum**, int **cmd**, [union semun **arg**]) предназначен для управления набором семафоров.

* где **semid** - идентификатор набора семафоров;
* **semnum** - номер семафора в наборе, определяющий объект, над которым выполняется управляющее действие, задаваемое значением третьего аргумента cmd; если объектом является набор, значение **semnum** — игнорируется;
* **cmd** - набор команд выполняемых на семафорами;
* для некоторых команд используется четвертый аргумент **arg**;



Далее рассмотрим наиболее важные **значения cmd**:

* Одиночные операции над семафором:
  + GETVAL - получить значение семафора и выдать его в качестве результата;
  + SETVAL - установить значение семафора равным arg.val.
* Групповые операции над набором семафоров:
  + GETALL - прочитать значения всех семафоров набора и поместить их в массив arg.array;
  + SETALL - установить значения всех семафоров набора равными значениям элементов массива.
* Информационные операции:
  + GETPID - узнать PID процесса, выполнившего последнюю операцию над семафором;
  + GETNCNT/GETZCNT — узнать число процессов, ожидающих увеличения /обнуления значения семафора.
* Управляющие команды над семафорами:
  + IPC\_STAT - получить информацию о состоянии набора семафоров ;
  + IPC\_SET - переустановить характеристики;
  + IPC\_RMID - удалить набор семафоров.

### **85. Задача об обедающих философах.**

«**Задачи об обедающих философах**» - основные этапы подготовки и реализации проекта по синхронизации процессов, которые могут приводить к взаимным блокировкам.

Классический вариант «**Задачи об обедающих философах**»:

* За круглым столом сидит несколько философов.
* В каждый момент времени каждый из них либо беседует, либо ест.
* Для процесса еды одновременно требуются две вилки. Поэтому, прежде чем в очередной раз перейти от беседы к приему пищи, философу необходимо дождаться, пока освободятся обе вилки - слева и справа от него, и взять их в руки.
* Немного поев, философ кладет вилки на стол и вновь присоединяется к беседе.
* Требуется разработать программную модель обеда философов.

**Главная проблема** в этой задаче, - **корректная дисциплина захвата и освобождения вилок**, иначе если каждый из философов, одновременно с другими, возьмется за вилку, лежащую слева от него, и будет ждать освобождения правой, то - обед не завершится никогда.

Одним из общих подходов решения задач синхронизации множества асинхронно взаимодействующих процессов является разделение структуры всего алгоритма на **два уровня иерархии**:

* первый — **нижний уровень** реализует сами действующие процессы, выделяя те общие ресурсы, которые нужны ему для обеспечения функционирования;
* второй — **верхний уровень** (программа-монитор) обеспечивает нижний уровень необходимыми данными, проводит первоначальную подготовку разделяемых ресурсов и, по возможности, устраняет возникающие проблемы синхронизации процессов нижнего уровня.

Для нашего случая:

* **нижний уровень** — программы процессов-философов, которые борятся за ресурс в виде двух вилок;
* **верхний уровень** — программа-монитор, которая подготавливает набор семафоров для синхронизации всех процессов-философов, создает и запускает эти процессы и отслеживает их завершение.

Каждый философ ведет себя независимо от других

* время **одной порции разговора** философа - случайное значение trnd;
* время **одной порции поедания** обеда философа - случайное значение ernd;
* общее **время поедания всего обеда** — ограничено, поэтому философ может завершить обед в несколько приемов.

Описание действий философа:

* философ какое-то время беседует (trnd), затем пытается взять вилки слева и справа от себя;
* когда ему это удается, он некоторое время ест (ernd), после чего освобождает вилки;
* так продолжается до тех пор, пока не будет съеден весь обед, после чего программа, моделирующая действия философа, закончит свою работу.

Для нормальной работы философа необходимы два ресурса, которые обеспечиваются двумя семафорами.

Чтобы реализовать корректную модель программы-философа, необходимо уточнить исходные данные, передаваемые ей в качестве **аргументов**.

* argv[1] — идентификатор набора семафоров;
* argv[2] — общее число философов;
* argv[3] — порядковый номер конкретного философа за столом.

Алгоритм работы программы-монитора (**QPH** - количество философов):

* порождается набор семафоров QPH: по одному семафору на каждую вилку;
* устанавливаются начальные значения семафоров: занятой вилке будет соответствовать значение 0, свободной — 1;
* запускаются QPH процессов, каждый из которых представляет одного философа, передавая им в качестве аргументов: идентификатор набора семафоров, общее количество всех философов и место конкретного философа за столом: философы нумеруются от 1 до QPH;
* ожидается, завершение работы всех процессов: когда все философы съедят свой обед;
* удаляется весь набор семафоров.

### **86. Прикладные средства пакета IPC.**

Поскольку ПО ЭВМ предназначено для решения прикладных задач, то разрабатывая и реализовывая алгоритм решения конкретной задачи, прикладной программист стремится сделать ее наиболее надежной, привлекательной и быстродействующей. Он стремится использовать наиболее эффективные системные средства ОС, которые бы помогли ему достичь желаемой цели.

Наиболее быстродействующими и эффективными средством являются системные вызовы ОС. Но системные вызовы требуют излишней детализации приложения. Поэтому был разработан пакет **System V IPC**, включающий единые средства **идентификации** (адресации), а также универсальное средство **синхронизации**, поддерживающее функции **семафоров**, первоначально разработанные Дейкстрой.

Дополнительно, чтобы освободить программиста от рутинной работы, связанной с «изобретением» индивидуальных средств передачи сообщений, а также для создания инструментов работы с данными, были разработаны средства: **разделяемой памяти** и **очередей сообщений**.

**Разделяемая память** — информационный объект данных, создаваемый и хранящийся в ядре ОС, который процесс может:

* создать или удалить;
* подключить к своему пространству данных или отсоединиться от него;
* работать с ним как с собственной структурой данных.

**Очередь сообщений** — универсальный «механизм» временного хранения в ядре ОС последовательности типизированных данных, которые процессы могут помещать и извлекать для своих нужд.

В совокупности с универсальным «механизмом» **адресации** (идентификации) и «механизмом» **семафоров**, **разделяемая память** и **очереди сообщений** образуют набор прикладных средств системного пакета **System V IPC**.

### **87. Разделяемая память.**

**Разделяемая память** — информационный объект данных, создаваемый и хранящийся в ядре ОС, который процесс может:

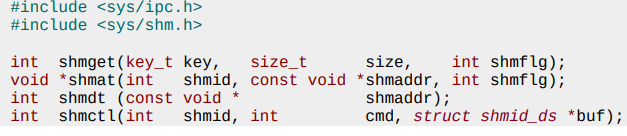
* создать или удалить;
* подключить к своему пространству данных или отсоединиться от него;
* работать с ним как с собственной структурой данных.

**Разделяемый объект памяти** - это объект, представляющий собой память ЭВМ, который может быть параллельно отображен в адресное пространство более чем одного процесса.

Таким образом, процессы могут иметь общие области виртуальной памяти и разделять содержащиеся в них данные.

**Единицей разделяемой памяти**- это сегмент, который может быть создан одним из асинхронно взаимодействующих процессов. Такой сегмент продолжает существовать, пока один из процессов не удалит его или ядро ОС не будет перезапущено.

**Создаваемый сегмент памяти** имеет атрибуты идентификации (адресации), которые являются общими для всех средств пакета IPC, а также индивидуальные права доступа, которые обеспечиваются каждым процессом, подключающим этот сегмент. Общий список системных вызовов, обеспечивающих работу с разделяемыми сегментами памяти, имеет вид:



Системным вызовом **shmget(...)** - отвечает за создание нового или получение идентификатора уже существующего разделяемого сегмента памяти (выше пример)

**key** — ключ доступа к средствам пакета IPC,

**size** — задает требуемый размер сегмента в байтах

**shmflg** – флаги, которые играют роль только при создании нового сегмента разделяемой памяти и определяют права различных пользователей при доступе к этому сегменту

При удачном завершении вызова shmget(...) возвращается идентификатор сегмента **shmid**, и -1 при ошибке,там еще error и он имеет множество значений в зависимости от ошибки (

Уже созданный сегмент разделяемой памяти пристыковывается к адресному пространству процесса с помощью системного вызова **shmid** - задает идентификатор разделяемого сегмента, полученный с помощью функции shmget(...)(пример выше)

**shmaddr** - адрес, по которому сегмент должен быть присоединен.

**shmflg** — флаг, определяющий свойства разделяемого сегмента памяти:

При удачном выполнении, системный вызов **shmat**(...) обновляет содержимое структуры shmid\_ds, связанной с разделяемым сегментом памяти, следующим образом:

* shm\_atime устанавливается в текущее время;
* shm\_lpid устанавливается в идентификатор вызывающего процесса;
* shm\_nattch увеличивается на 1.

В случае ошибки функция **shmat**(...) возвращает значение -1, а переменной errno присваивается номер ошибки:

Отсоединение, ранее подключенного сегмента разделяемой памяти, осуществляется системным вызовом **shmaddr** - задает начальный адрес отсоединяемого сегмента.(пример выше)

При успешном завершении функции возвращется результат равеный 0 и обновляется содержимое структуры shmid\_ds:

* shm\_dtime устанавливается в текущее время;
* shm\_lpid устанавливается в идентификатор вызывающего процесса;
* shm\_nattch уменьшается на 1; если это значение становится равным 0, а сегмент помечен на удаление, то сегмент удаляется из памяти; в любом случае, освобождается занятая ранее этим сегментом область памяти в адресном пространстве процесса.

В случае неудачи, возвращается -1 и в переменную errno устанавливается код ошибки:

Общее управление сегментами разделяемой памяти, с использованием структуры типа shmid\_ds, осуществляется посредством системного вызова: **shmid** - является дескриптором System V IPC для сегмента разделяемой памяти или - значением, которое вернул системный вызов shmget(...), при создании сегмента или при его поиске по ключу.;

При удачном выполнении возвращается 0, а при ошибке -1, причем переменной errno присваиваются следующие значения:

### **88. Задача о читателях и писателях.**

Здесь мы используем аппарат разделяемых сегментов памяти предоставляет нескольким процессам возможность одновременного доступа к общей области памяти.

В качестве средства синхронизации используются семафоры

Чтобы правильно использовать семафоры, при доступе к разделяемым сегментам памяти, необходимо:

* тщательно проанализировать задачу и выделить в процессах критические интервалы (области программы);
* определить механизмы, обеспечивающие взаимное исключение разделяющих общие данные процессов

Алгоритм процесса-читателя, представленный, :

* создаются, если не созданы, ключи key1 и key2, массив семафоров sembuf[2] и сегмент разделяемой памяти shareseg;
* процесс-читатель, в бесконечном цикле, через случайный интервал времени tsleep порождает дочерние процессы; 16
* каждый дочерний процесс: ожидает возможности чтения, а затем — завершает работу, если shareseg=0, или читает случайное время tread, если shareseg>0.

### **89. Передача сообщений.**

3 средством, входящим в System V IPC, являются очереди сообщений.

**Очереди сообщений** - это наиболее семантически нагруженный способ взаимодействия процессов через каналы связи, в котором на передаваемую информацию накладывается определенная структура, так что процесс, принимающий данные, может четко определить, где заканчивается одна порция информации и начинается другая.

Такая модель позволяет задействовать один и тот же канал связи для передачи данных в двух направлениях между несколькими процессами.

Очереди сообщений, как семафоры и разделяемая память, являются средством связи с непрямой адресацией, что требует инициализацию и специальные действия

**Пространством имен** очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции ftok(), а для передачи данных используются системные примитивы, в виде функций send() и receive(), которым в качестве параметра передаются IPC-дескрипторы очередей сообщений, однозначно идентифицирующие эти данные во всей вычислительной системе.

**Очереди сообщений** имеют следующие особенности:

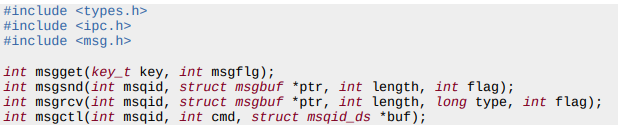
* располагаются в адресном пространстве ядра операционной системы в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди.
* каждый элемент списка представляет собой отдельное сообщение.
* каждое сообщение имеет атрибут, называемый типом сообщения.

**Выборка сообщений** из очереди, соответствующая примитиву receive(), может быть выполнена тремя способами:

* В порядке FIFO, независимо от типа сообщения.
* В порядке FIFO, для сообщений конкретного типа.
* Первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

Реализация примитивов **send() и receive()** обеспечивает скрытое от пользователя взаимоисключение, во время помещения сообщения в очередь или его получения из очереди

Общий набор примитивов передачи сообщений представлен четырьмя системными вызовами



Системный вызов **msgget()** предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этой очереди: целое неотрицательное число, однозначно характеризующее очередь сообщений внутри вычислительной системы и использующееся в дальнейшем для других операций с ней.

Параметр **msgflg** – имеет значение только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией, с помощью операции побитовое или – " | ", следующих предопределенных значений и восьмеричных прав доступа:

Системный вызов **msgget**() возвращает значение дескриптора System V IPC для очереди сообщений, или -1 при ошибки

Системный вызов **msgsnd**() предназначен для помещения сообщения в очередь сообщений.

Системный вызов **msgrcv**() предназначен для получения сообщения из очереди сообщений.

### **90. Графические среды ОС.**

Многие системы на Linux не имеют встроенного графического редактора и поэтому они используют набор приложений, взаимодействующих на основе модели «клиент/сервер» в нашем случае **X Window System**

**X Window** - сложная система, подробно описаная большим количеством первоисточников. Основой ее является программа X-сервер, которая через драйверы устройств взимодействует с видеокартой ЭВМ, клавиатурой, мышью и монитором компьютера.

Именно Х-сервер:

* устанавливает и переключает графические режимы видеокарты ЭВМ;
* рисует элементы изображений;
* определяет координаты мыши;
* формирует программные прерывания при нажатии кнопок мыши и клавиатуры.

Все остальные программы, включая менеджер окон, взаимодействуют с **Х-сервером** по особому протоколу, который называется **X-протокол**, или протокол сетевой связи (X Network Protocol).

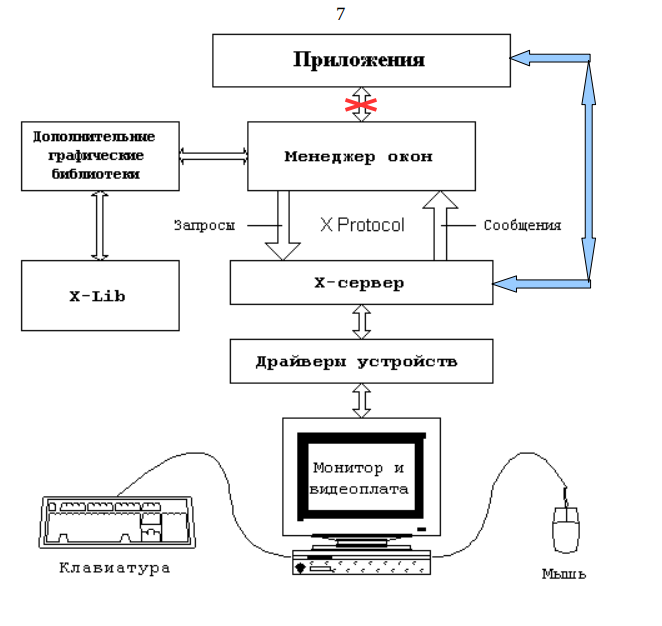
Для написания программ, поддерживающих Х-протокол, имеется базовая

библиотека X-lib . На основе этой библиотеки пишутся дополнительные графические библиотеки более высокого уровня.

Обычно, менеджеры окон, рабочие столы пользователей и сложные графические приложения пишутся с использованием этих библиотек.

**Общая архитектура X Window** может быть представлена рисунком ниже, где Х-сервер обрабатывает 4 типа сообщений:

* Запрос – клиент требует нарисовать что-либо в окне или запрашивает у сервера информацию;
* Ответ – сервер отвечает на запрос;
* Событие – сервер сообщает клиенту о событии, например, о нажатии клавиши пользователем;
* Ошибка – сервер сообщает об ошибке в запросе клиента.



### **91. Рабочий стол пользователя.**

Писать и настраивать приложения для Х-сервера довольно сложно поэтому все современные ОС, использующие графический интерфейс с пользователем, имеют специальное приложение, которое называется **оконный менеджер**

Оконный менеджер - это приложение, которое отвечает за размещение, декорирование окон и за взаимодействие между окнами:

* изменение размеров,
* максимизация,
* свертывание,
* закрытие.

Оконный менеджер взаимодействует с графическим сервером X11А он в свою очередь взаимодействует с видеокартой и с устройствами ввод\вывод (клава мышь монитор)

**Именно оконный менеджер** обеспечивает функциональность графической среды пользователя.(если чо это все рабочий стол)

Учебная система ОС УПК АСУ использует рабочий стол.

* Xfce — как окружение рабочего стола;
* xfwm4 — как оконный менеджер.

### **92. Различие графических сред ОС.**

Имеются операционные системы, которые не могут работать без графического окружения.

( К таким ОС относятся: MS Windows и MacOS.)

Для UNIX и Linux, графическое окружение ОС является необязательным и может быть отключено с целью экономии ресурсов ЭВМ.

В целом, оконные менеджеры могут работать:

* как вместе с окружением рабочего стола;
* так и отдельно от него.

Функционал, предоставляемый оконными менеджерами, может достаточно сильно различаться.Деляться на тайловые(окна не перекрываются) и композитные(окна перекрываться.).

### **93. X-сервер UNIX.**

Важной особенностью **Х-сервера** является возможность его работы на стеке протоколов TCP/IP с программами, запущенными на удаленных компьютера.

Обычно, для этих целей используется ассинхронная связь (протокол UDP)

Кроме того, на компьютере может быть запущено несколько Х-серверов, которые выводят графическую информацию на разные дисплеи

**X-сервер** запускается дисплейным менеджером, который предлагает клиенту ОС ввести login и password для открытия сессии и входа в нее

В используемом дистрибутиве **ОС УПК АС**У, все необходимые настройки Х11 уже проведены, поэтому Х-сервер уже готов к работе в сети.

Обычно используют только 1 X- сервер Для соединения он прослушивает некоторый порт по умолчанию это порт с номером 6000 .

Чтобы прикладные программы знали на каком компьютере и на каком дисплее находится Х-сервер, используется:

* переменная окружения DISPLAY;
* аргумент программы с ключем: -display

Чтобы разрешить **Х-серверу** подключение программ, используется утилита **xhost**. Если запустить **xhost** без параметров, то мы получим сообщение, что только авторизованный пользователь может быть подключен.

Если нужно разрешить подключение с компьютера по известному адресу, например, 192.168.0.12, то нужно использовать команду: sudo xhost +192.168.0.12

Символы + - означают разрешение или запрет подключения.

### **94. Архитектура шины D-Bus.**

D-Bus – еще одна система межпроцессного взаимодействия (Interprocess Communication или IPC)

Таким образом, D-Bus - это система межпроцессного взаимодействия, предоставляющая приложениям несколько шин для передачи сообщений и обеспечивающая беспроблемную связь десктопных приложений. Причем, поддерживается не только широковещательная рассылка сообщений (сигналов), но и удаленный вызов методов

Основные понятия шины D-Bus:

• Системная шина (System bus) — данная шина создается при старте сервера D-Bus. К ней подключены системные сервисы: HAL, NetworkManager, bluez, WPA Supplicant и другие. На этой шине работают сервисы, которые нельзя отнести к какой-то определенной пользовательской сессии, и которые относятся к системе в целом.

• Сессионная шина (Session bus) — данная шина создается на каждый вход (login) пользователя в систему. К данной шине подключаются приложения пользователя, прошедшего процедуру login. Также через нее проходит общение программ запущенных в данной рабочей сессии пользователя.

• Имя на шине (Bus name). Каждая программа, подключенная к шине, получает свое уникальное имя, которое начинается с двоеточия (":") и представлено двумя числами, разделенными точками. Например, «:1.5».

• Имя сервиса (Service name). В D-Bus — приложение может взять несколько дополнительных имен, чтобы другие программы могли с ней связываться. Имя сервиса представляет собой набор из символов, разделенных точками ("."). Пример: сервис имеет реальное имя ":1.5", и символьное представление - "org.freedesktop.NetworkManager". Мы можем соединяться как с ":1.5", так и с "org.freedesktop.NetworkManager". Это - один и тот же сервис. Нужно ли создавать символьное представление имени клиент решает самостоятельно.

• Путь к объекту (Object path). Путь к некоторому объекту внутри адресуемого сервиса: "/org/freedesktop/NetworkManager/Device/eth0". Сервис может обслуживать несколько объектов.

• Интерфейс (Interface) — каждый объект предоставляет доступ к некоторому набору методов и сигналов, каждый из которых называется интерфейсом. При этом, один объект может предоставлять несколько разных интерфейсов, например: "org.freedesktop.NetworkManager.Device.Wireless".

• Метод (Method) — некоторое действие, которое может производить объект в данной программе и на данном интерфейсе. Аналогично функциям на языке С, метод может вернуть: набор некоторых данных, код ошибки, либо может вообще не возвращать данных (void).

• Сигнал (Signal) — некоторое сообщение, которое распространяется среди всех программ, подписанных на этот сигнал на этом интерфейсе данного объекта и данной программы. Сигналы могут содержать набор данных.

• Сообщение (Message) — каждая передача данных на шине представляется в виде сообщений. Они могут быть 4-х типов: вызовы методов, сигналы, результаты методов, ошибки.

• Прокси-объект (Proxy object) - объект одного из языков программирования: C++, Python, Java и других, вызовы методов которых проецируются на вызовы методов шины D-Bu

Схема взаимодействия



**Взаимодействие** между приложениями происходит через серверное приложение Bus Daemon Process, которое реализовано в виде программы dbus-daemon.

**Процесс взаимодействия** обеспечивается специальным механизмом сокетов: Socket Bidirectional Message Stream.

**Точками соединения** являются универсальные программные конструкции, которые обозначены как DbusConnectionInstance.

**Bus Daemon Process обеспечивает** диспетчеризацию именованных сообщений между приложениями, используя системные вызовы, реализованные в библиотеке libdbus.

**Взаимодействующие приложения** используют всего два типа вызовов: Incoming Call и Outgoing Call

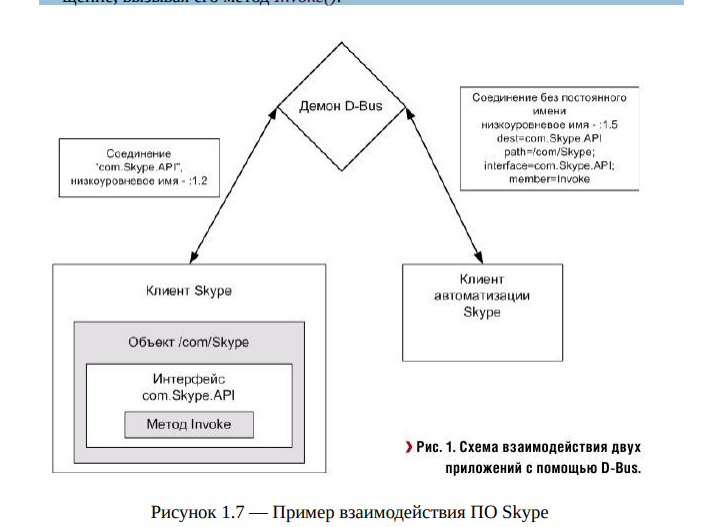
### **95. Библиотека libdbus.**

**Низкоуровневое взаимодействие** приложений через шину D-Bus обеспечивается с помощью библиотеки **libdbus**

**Простейший пример** приложений, взаимодействующих с помощью библиотеки libdbus, рассмотрим для случая ПО Skype.

• запущено приложение Skype, которое регистрируется на шине с дополнительным именем com.Skype.API;

• другое приложение, «Клиент автоматизации Skype», посылает первому сообщение, вызывая его метод Invoke().

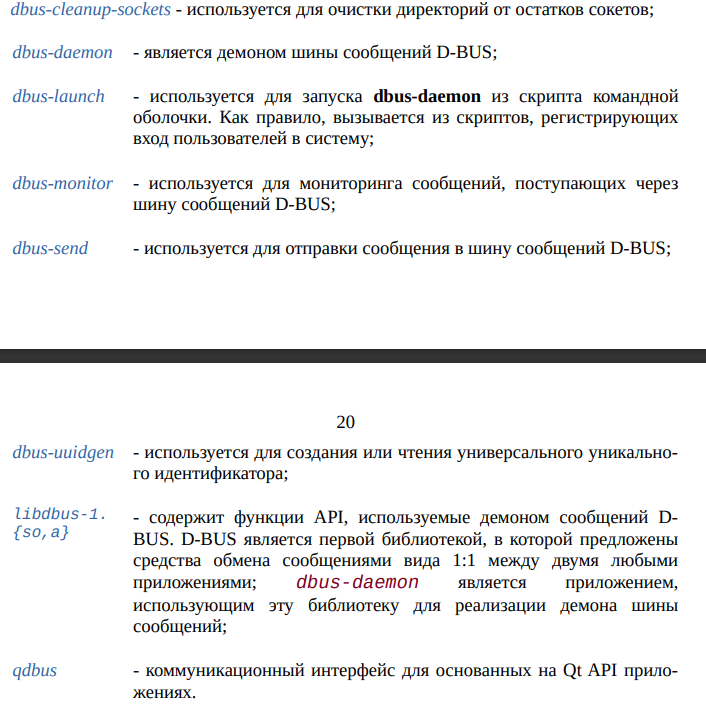


### **96. Проекции ПО D-Bus на языки программирования.**

Для эффективного использования возможностей шины применяются проекции libdbus, реализованные на различные языки.

Широко известны следующие проекции: GLib API ,Python API,Qt API

Для доступа к шине D-Bus из языка shell используется набор утилит, которые лежат в руководстве man:



Если вкратце для работы с D-Bus необходимо хорошо знать приложение, с которым осуществляется взаимодействие, или написать такое приложение самому.