МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ
Зав. кафедрой АСУ, профессор

А.М. Кориков

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебно-методическое пособие

для студентов уровня основной образовательной программы: бакалавриат
направление подготовки: **09.03.01 - Информатика и вычислительная техника**
направление подготовки: **09.03.03 - Прикладная информатика**

Разработчик
доцент кафедры АСУ

В.Г. Резник

2016

Резник В.Г.

Операционные системы. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 183 с.

Учебно-методическое пособие предназначено для изучения теоретических вопросов и выполнения лабораторных работ по дисциплине «Операционные системы» для студентов кафедры АСУ ТУСУР уровня основной образовательной программы бакалавриат направлений подготовки: «09.03.01 - Информатика и вычислительная техника» и «09.03.03 - Прикладная информатика».

Оглавление

Введение

Дисциплина «*Операционные системы* (ОС)» изучается студентами кафедры АСУ ТУСУР уровня основной образовательной программы бакалавриат на третьем курсе обучения.

Объем изложенного учебного материала соответствует:

- первой части обучения для направления подготовки: «09.03.01 - Информатика и вычислительная техника»;
- полному курсу обучения для напрвления подготовки: «09.03.03 - Прикладная информатика».

Целью дисциплины является обучение студентов основным понятиям и базовым концепциям, положенным в архитектуры современных операционных систем, а также приобретение студентами практических навыков, необходимых для успешного использования полученных знаний.

Указанная цель достигается комплексной методикой проведения учебных занятий, основанной на:

- модульном построении учебного материала данного пособия, согласованного по изложению теоретической части отдельных разделов дисциплины и проведению соотвествующих лабораторных работ;
- учебным программным комплексом каферды АСУ, обеспечивающим учебный материал данного пособия вычислительными и программыми ресурсами для проведения лабораторных занятий.

Формальной и базовой основой изложенного учебного материала являются:

- научное издание Таненбаума Э. [1];
- учебник для вузов Синицына С.В. [2];
- учебно-методическое пособие Резник В.Г. [3], доступное по электронному адресу: http://asu.tusur.ru/learning/books/b13.pdf.

Методика проведения процесса обучения по данному курсу предполагает использование учебных классов кафедры АСУ, которые:

- оборудованны проекторами для демонстрации теоретического материала;
- имеют персональную вычислительную технику с установленной ОС УПК АСУ для проведения лабораторных работ.

1 Тема 1. Назначение и функции ОС

Кроме наиболее широко известных ОС – **MS Windows** и **Linux**, имеется целый ряд весьма интересных и распространенных линий развития ОС. Это - прежде всего операционные системы крупных фирм:

- **IBM** – в 1960-х – 1970-х годах разработала OS IBM 360/370; затем – ОС для персональных компьютеров OS/2; в настоящее время наиболее современными ОС этой фирмы являются ***z/OS*** и ***z/VM***;
- **Apple** – с начала 1980-х годов развивает семейство ОС ***MacOS***, которые характеризуются улучшенным графическим пользовательским интерфейсом;
- **Oracle/Sun** – с начала 1980-х гг. фирма Sun развивает ОС ***Solaris*** (диалект UNIX);
- **Hewlett-Packard** – развивает собственный диалект UNIX – систему ***HP/UX***;
- **Novell** – одна из ведущих фирм в области сетевых технологий; развивает семейство сетевых ОС ***NetWare***; в настоящее время - ***Open Enterprise Server*** (сетевая ОС, включающая все сетевые возможности NetWare и возможности распространенного диалекта Linux - ***openSUSE***).

Это далеко не полный перечень коммерческих и исследовательских ОС, включающий сотни наименований.

В данной теме рассматривается ряд концепций и представлений, которые сформировались в процессе становления и развития дисциплины «*Операционные системы* (ОС)». Хотя отдельные такие представления отражают разные взгляды, сформулированные в различных терминах, их общий набор должен сформировать целостный образ предмета изучения и обеспечить студента базовыми понятиями, необходимыми для дальнейшего более подробного изучения материала.

1.1 ОС как базовая часть систем обработки данных (СОД)

Прежде чем изучать ОС, необходимо ответить на вопрос: «*Зачем нужны компьютеры?*».
Одним из возможных ответов: «*Компьютеры нужны для обработки данных*».
Известный русский ученый, Ларионов А.М., анализируя возможности обработки данных на ЭВМ, дает определение и классификацию различных систем обработки данных.

Система обработки данных (СОД) — совокупность технических средств и программного обеспечения, предназначенная для информационного обслуживания пользователей и технических объектов.

Общая структура классификации СОД, данная Ларионовым А.М., приведена на рисунке 1.1. Ее с успехом можно использовать в качестве ориентира при изучении различных дисциплин, связанных с применением вычислительной техники.

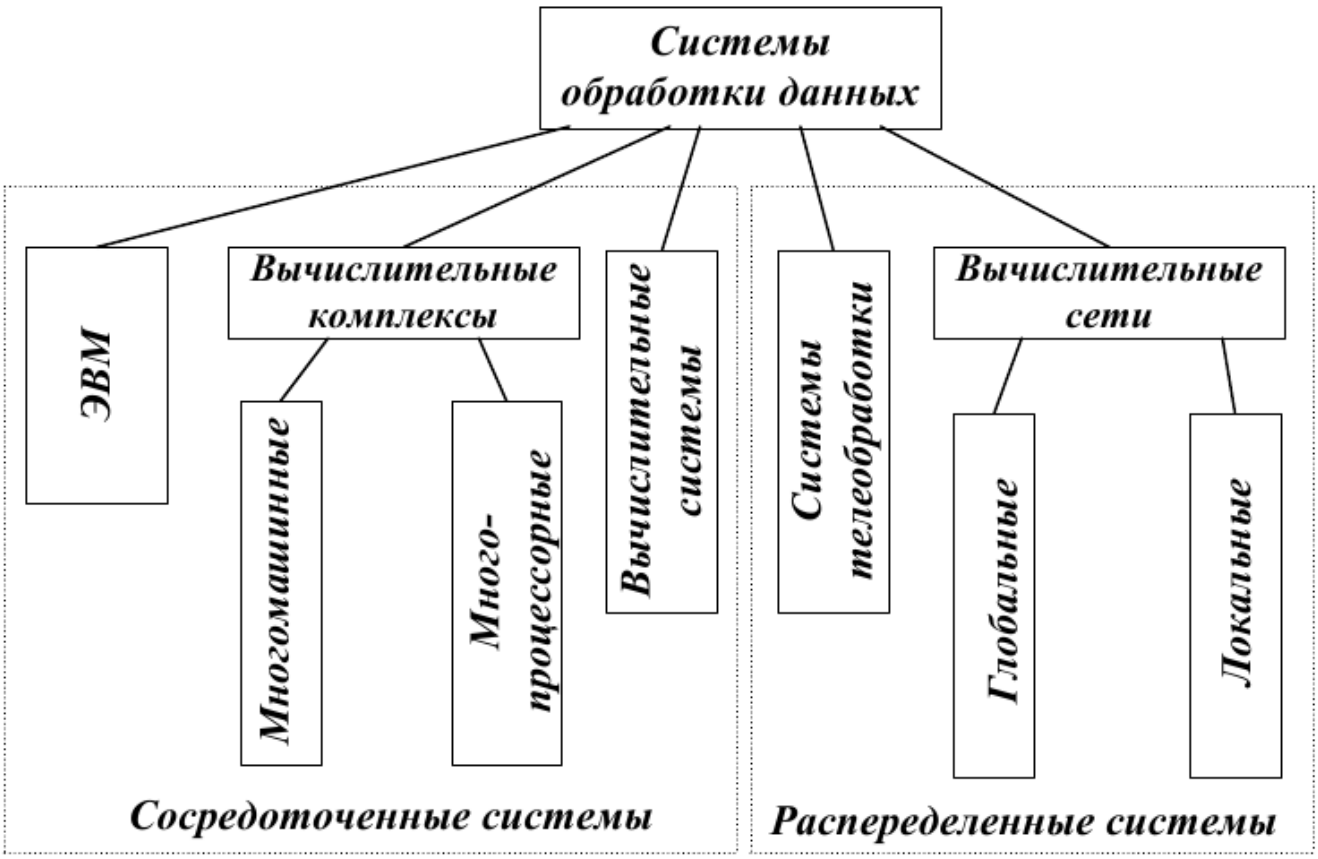

Рисунок 1.1 — Структура классификации СОД (Ларионов А.М.)

Хорошо видно, что вся классификация разделена на два больших класса:

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

Нетрудно догадаться, что каждый элемент такой классификации имеет свою ОС, хотя их функциональные возможности могут сильно отличаться.

С другой стороны, мы понимаем, что *ОС — это программное обеспечение (ПО)*, которое устанавливается на аппаратную часть вычислительной техники. С этой точки зрения, рассматривая отдельную ЭВМ, мы будем различать:

- *техническую часть* — аппаратное обеспечение компьютера, упрощенная архитектура которого представлена на рисунке 1.2;
- *программную часть* — программное обеспечение компьютера, общая классификация которого показана на рисунке 1.3.

Архитектура современного общедоступного компьютера представляет набор функциональных компонент, во многом работающих независимо друг от друга, но согла-

сованно взаимодействующих через общую системную манистраль.

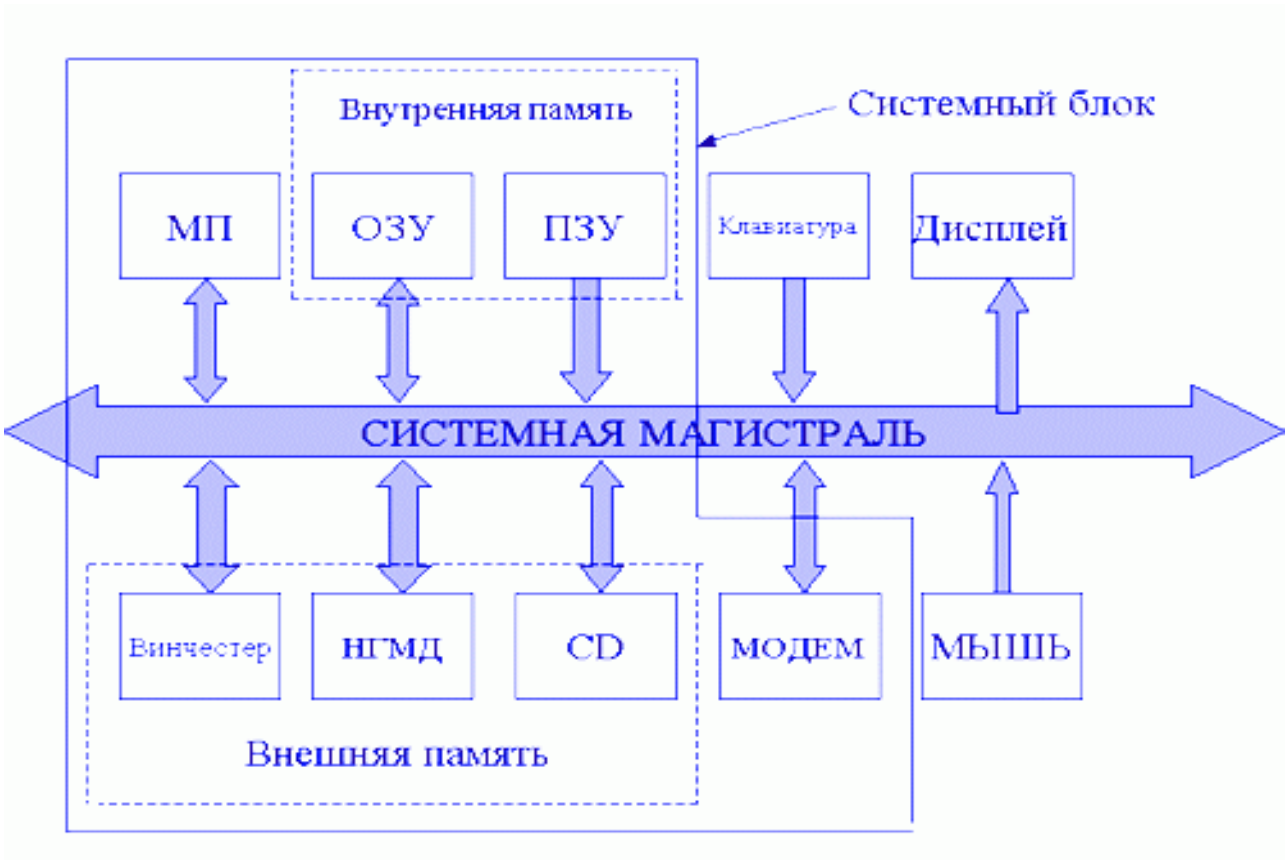

Рисунок 1.2 - Архитектура аппаратной части ЭВМ

Основу архитектуры аппаратной части ЭВМ составляет *системный блок*, в котором размещены:

1) микропроцессор (**МП**);
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 (**CD**) и дискет (**НГМД**).

В системном блоке находятся также *интерфейсные платы*: сетевая, видеопамяти, обработки звука, модем (модулятор-демодулятор), платы, обслуживающие устройства ввода-вывода: клавиатуры, дисплея, "мыши", принтера и другие.

Программная часть ЭВМ (рис. 1.3) условно разделяется на три 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.

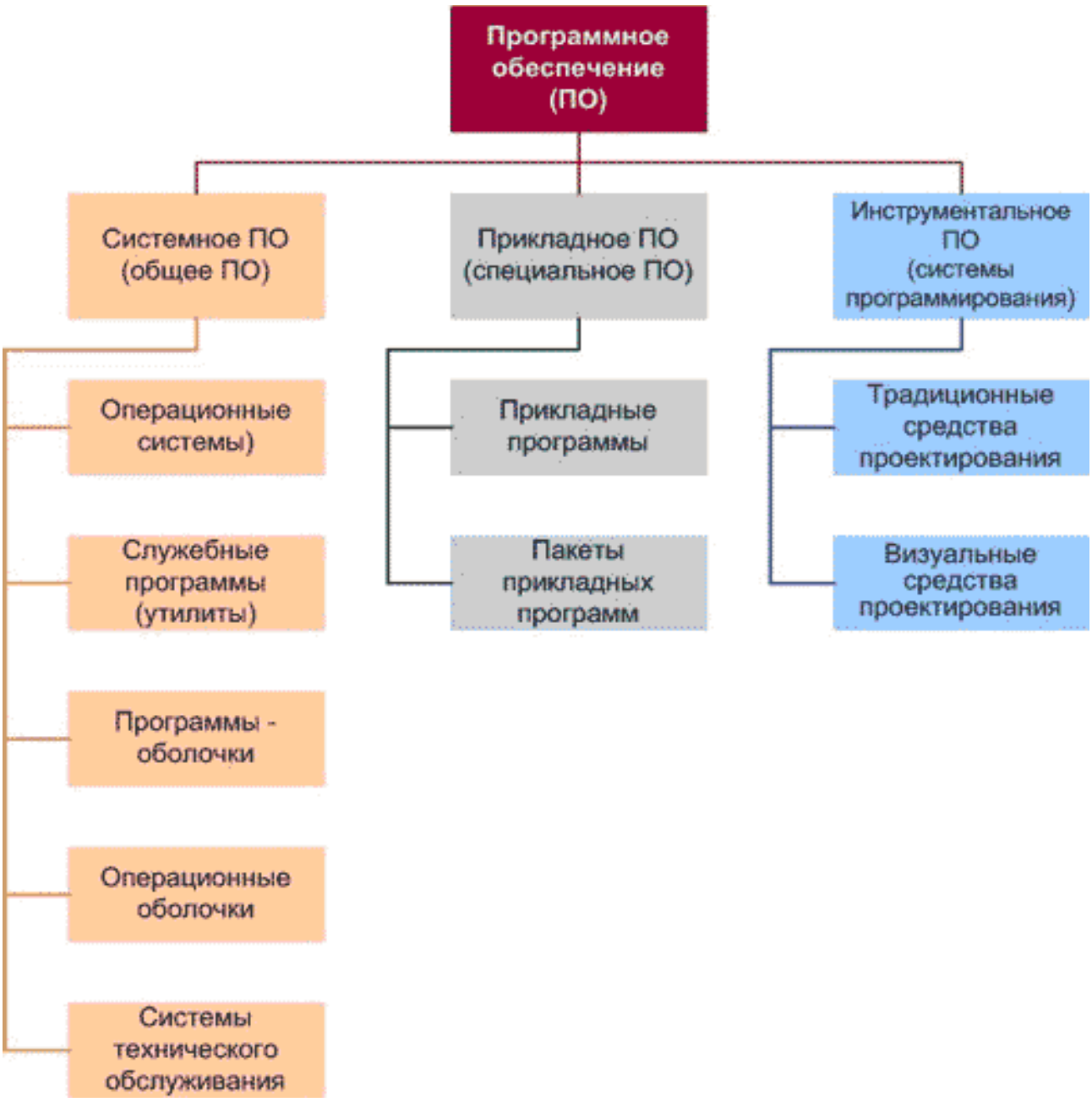

Рисунок 1.3 - Классификация ПО ЭВМ

Хорошо видно, предметом нашей дисциплины является *категория системного ПО*.

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

1. *операционные системы* — 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, например, Far, Total Commander;
3. *операционные оболочки* – интерфейсные системы, которые используются для создания графических интерфейсов, мультипрограммирования и другие;

4. *драйверы* - программы, предназначенные для управления портами периферийных устройств; обычно загружаются в оперативную память при запуске компьютера;
5. *утилиты* - вспомогательные или служебные программы, которые представляют пользователю ряд дополнительных услуг.

Замечание

Часто бывает сложно разделить ПО, относящееся к утилитам, и прикладное ПО, поскольку часть утилит входит в состав ОС, а другая часть поставляется и функционирует автономно.
Общепринято считать, что к утилитам относятся:

- *диспетчеры файлов* или *файловые менеджеры*;
- *средства динамического сжатия данных*, которые позволяют уменьшить размеры файлов и увеличить количество информации на диске за счет ее динамического сжатия;
- *средства просмотра* и *воспроизведения*;
- *средства диагностики* и *контроля*, которые позволяют проверить конфигура-цию компьютера и работоспособность его устройств, прежде всего «жестких дисков»;
- *средства коммуникаций* (коммуникационные программы), предназначенные для организации обмена информацией между компьютерами;
- *средства обеспечения компьютерной безопасности*: резервное копирование, антивирусное ПО и другие.

1.2 Серверные ОС и рабочие станции

Несмотря на кажущуюся простоту понятий *сервер* и *рабочая станция*, необходимо внимательно относиться к контексту, в котором эти термины употребляются. Прежде всего, следует уточнить идет ли речь об апаратном или о программном обеспечении.
В контексте аппаратного обеспечения ЭВМ:

- *сервер* — существительное от глагола *to serve* — служить; 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* — workstation — 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.

Фактически, сервер и рабочая станция могут иметь одинаковую аппаратную конфигурацию, что во многом ограничивает возможности данного контекста.

Наиболее широко термины сервер и рабочая станция используются в контексте программного обеспечения или в комбинации с котекстом 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*.

На ранней стадии развития, многие ОС не поддерживали работу в сети. Например, MS DOS и MS Windows, первоначально создавались как рабочие станции, предполагающие автономную работу ЭВМ. Со временем, такие ОС стали использовать сетевое ПО сторонних разработчиков, а те ОС, которые имели собственное сетевое ПО, стали называться ***сетевыми ОС***. В настоящее время, практически все ОС способны работать в сети, поэтому необходимость в дополнительной классификации отпала сама собой. Тем не менее, градация ЭВМ осталась, но перешла в область системного ПО и дистрибутивов 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*.

Замечание

Применительно к ограничениям уровня изучения нашей дисциплины, различия между серверами и рабочими станциями являются не существенными, поскольку определяются ***специализацией системного и прикладного ПО ЭВМ***. ПО ОС УПК АСУ создано на основе дистрибутива типа *desktop*, поэтому для работы в качестве сервера требуется установка дополнительного ПО.

В целом, объем нашего курса предполагает изучение ПО ОС, *на уровне desktop*, *ограниченной отдельной ЭВМ*.

1.3 Многослойная структура ОС

Первоначально, *ОС были монолитными и не имели архитектуры*. Например, корпорация IBM в 1964 году стала разрабатывать первую версию ОС OS/360 и за 5 лет, коллектив из 5000 человек написал более 1 млн строк кода. Постепенно стало ясно, что разработка ОС должна:

- *вестись* на основе модульного программирования;
- *иметь* иерархическую структуру.

В результате, были разработаны концептуальные требования к архитектуре 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 другими.

В концепции многоуровневой (*многослойной*) иерархической машины, структура ОС представляется рядом слоев, показанных на рисунке 1.4. Здесь, каждый внутренний слой обслуживает вышележащий слой через *межслойный интерфейс*.

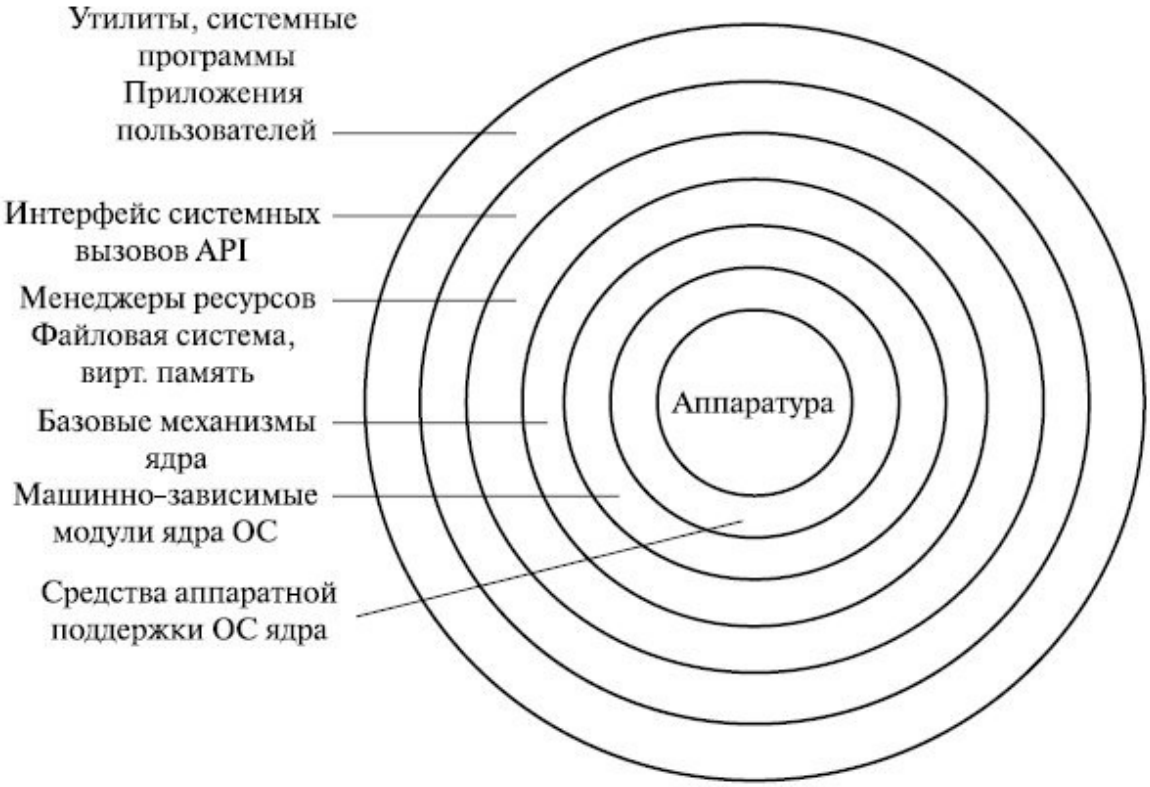

Рисунок 1.4 - Иерархическая архитектура ОС

Такая организация системы существенно упрощает ее разработку. Она позволяет:

- сначала, "*сверху вниз*" определить функции слоев и межслойные интерфейсы;
- при детальной реализации, двигаясь "*снизу вверх*", – можно наращивать мощность функций слоев;
- модули каждого слоя можно изменять без необходимости изменений в других слоях, но *не меняя межслойных интерфейсов*!

В общем случае, повышение устойчивости работы ОС обеспечивается переходом ядра ОС в *привилегированный режим*.

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

1.4 ОС как базовая часть ПО ЭВМ

Из рисунка 1.4 хорошо видно, что, в классической архитектуре ОС, прикладным программам пользователей отводится верхний, последний слой. Все остальные слои архитектуры составляют *ядро ОС*, которое непосредственно взаимодействует с аппаратным обеспечением ЭВМ.

Ядро ОС составляет сердцевину 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*.

Особый класс функций ядра служит для поддержки приложений, создавая для них так называемую *прикладную программную среду*. Все приложения обращаются к ядру со специальными запросами – *системными вызовами*. Примеры системных вызовов:

- открытие и чтение файла,
- получение системного времени,
- вывод информации на дисплей компьютера и другие.

Функции ядра, которые могут вызываться приложениями, образуют *интерфейс прикладного программирования* – **API** (*Application Programming Interface*).

Хотя архитектура аппаратной части многих ЭВМ может быть адекватно представлена рисунком 1.2, конкретные реализации такой архитектуры могут быть различны:

- *различаются* процессора и поддерживаемый набор команд;
- *различаются* шины компьютера и устройства подключения к ним;
- *постоянно идет развитие* и *изменение* конструктивных особенностей всех внешних устройств.

В такой ситуации, *каждое ядро ОС* реализует некоторую абстрактную и более упрощенную архитектуру ЭВМ, доступ которой реализуется через стандартный набор функций, называемый *функциями системных вызовов*.
В результате, прикладной программист рассматривает ядро ОС как некоторую *абстрактную (**виртуальную**) машину*, которая является средой для выполнения его программ.

Систематизируя различные системные вызовы и развивая идею виртуальной машины, мы с точностью до терминологии можем утверждать, что каждое ядро ОС, абстрагирует три базовых концепции: ***файл***, ***пользователь*** и ***процесс***. Но прежде чем обосновать это, рассмотрим более подробно взаимодействие прикладных программ пользователя с защищенным ядром ОС, через интерфейс API. Кроме того, следует также учесть, что на основании рисунка 1.4 можно формировать различные типы ядер ОС.

1.5 Режимы ядра и пользователя

Чтобы повысить надежность работы ОС, ее ядро работает в специальном *привилегированном (защищенном) режиме*. Соответственно, режим, в котором работают утилиты и остальное прикладное ПО ОС называется *режимом пользователя*.

Сначала рассмотрим взаимодействие ПО ЭВМ для ***классической архитектуры ядра ОС UNIX***, которое показано на рисунке 1.5.

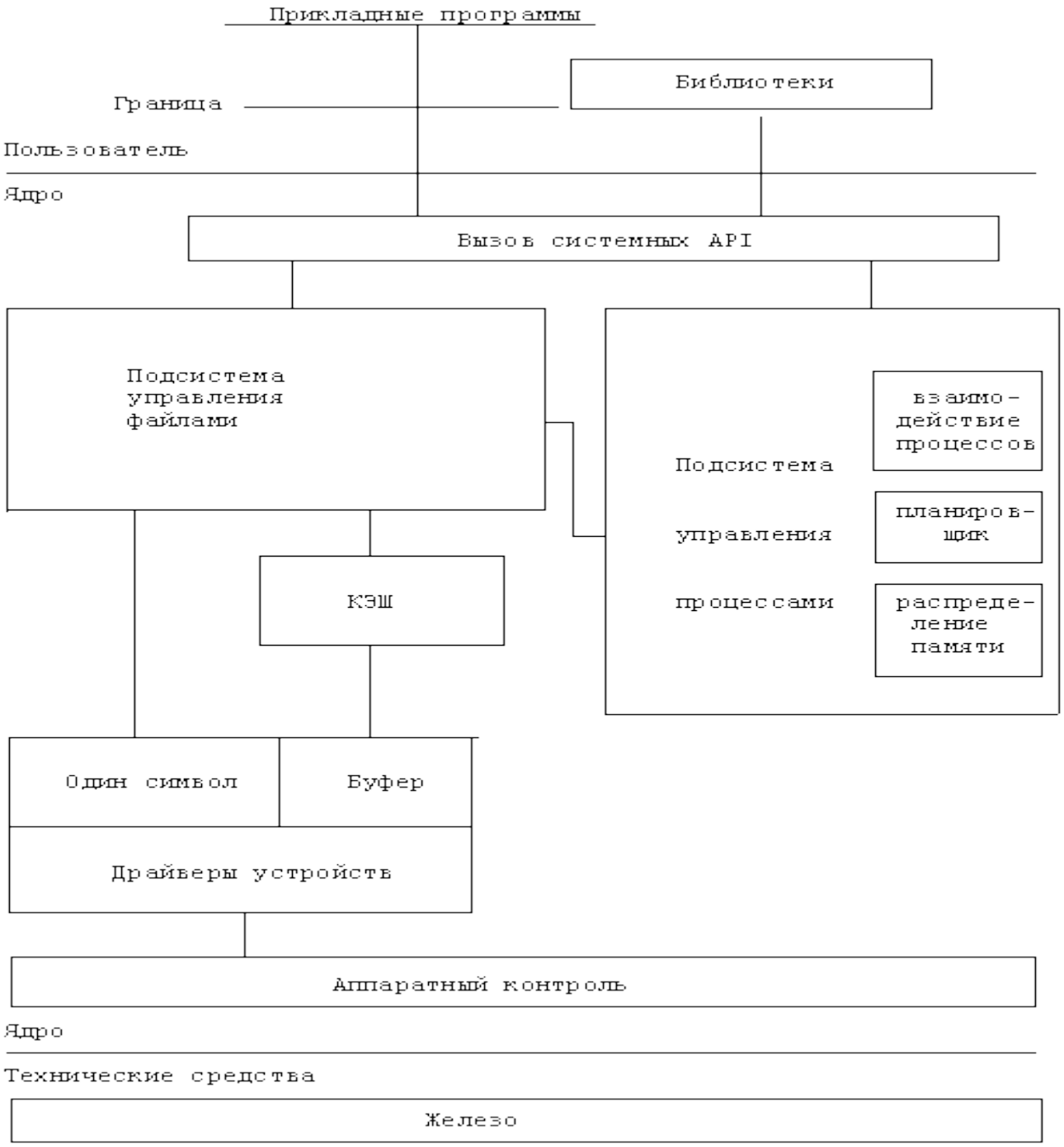

Рисунок 1.5 - Классическая архитектура ПО ОС UNIX

Классическая архитектура ПО ОС UNIX содержит *монолитное ядро ОС*, которое согласно рисунку 1.4 содержит ПО всех уровней, кроме последнего. Системный вызовов такого ядра ОС происходит в два этапа и показан на рисунке 1.6:

- *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. Однако такое решение стало классическим и используется во многих ОС: UNIX, VAX, VMS, IBM OS/390, OS/2 и других.

Рисунок 1.6 - Переключение режимов монолитного ядра ОС

Замечание

Многослойная классическая многоуровневая архитектура ОС не лишена своих проблем:

- *значительные изменения одного из уровней* могут иметь трудно предвидимое влияние на смежные уровни;
- *многочисленные взаимодействия между соседними уровнями* усложняют обеспечение безопасности работы ОС.

Альтернативой классическому варианту архитектуры ОС является *микроядерная архитектура ОС*.

Суть этой архитектуры состоит в следующем:

- В привилегированном режиме остается работать только очень небольшая часть ОС, называемая *микроядром*.
- *Микроядро защищено* от остальных частей ОС и приложений.
- В состав микроядра входят *машинно-зависимые модули*, а также *модули, выполняющие базовые механизмы обычного ядра*.
- Все остальные, более высокоуровневые функции ядра, оформляются как *модули, работающие в пользовательском режиме*.

На рисунке 1.7 представлено сравнение классической и микроядерной архитектур ОС:

- *менеджеры ресурсов*, являющиеся неотъемлемой частью обычного ядра, становятся "периферийными" модулями, работающими в пользовательском режиме;
- *внешние*, по отношению к микроядру компоненты ОС, реализуются как обслуживающие процессы;
- *между собой* эти модули взаимодействуют как равноправные партнеры с помощью обмена сообщениями, которые передаются через микроядро.

Рисунок 1.7 - Сравнение классической и микроядерной архитектур ОС

При такой организации, *менеджеры ресурсов*, вынесенные в пользовательский режим, называются ***серверами ОС***. Схематично механизм обращений к функциям ОС, оформленным в виде серверов, выглядит, как показано на рисунке 1.8.

Для микроядра, схема смены режимов, при выполнении системного вызова в ОС, показана на рисунке 1.9. Хорошо видно, что выполнение системного вызова сопровождается *четырьмя переключениями режимов (4 t)*, а в классической архитектуре – *двумя (2 t)*. Следовательно, производительность ОС с микроядерной архитектурой, при прочих равных условиях, будет ниже, чем у ОС с классическим ядром.

Замечание

По многим литературным источникам, вопрос масштабов потери производительности в микроядерных ОС является спорным. *Большинство ядер дистрибутивов ОС Linux являются монолитными*.

Рисунок 1.8 - Системные вызовы через микроядро ОС

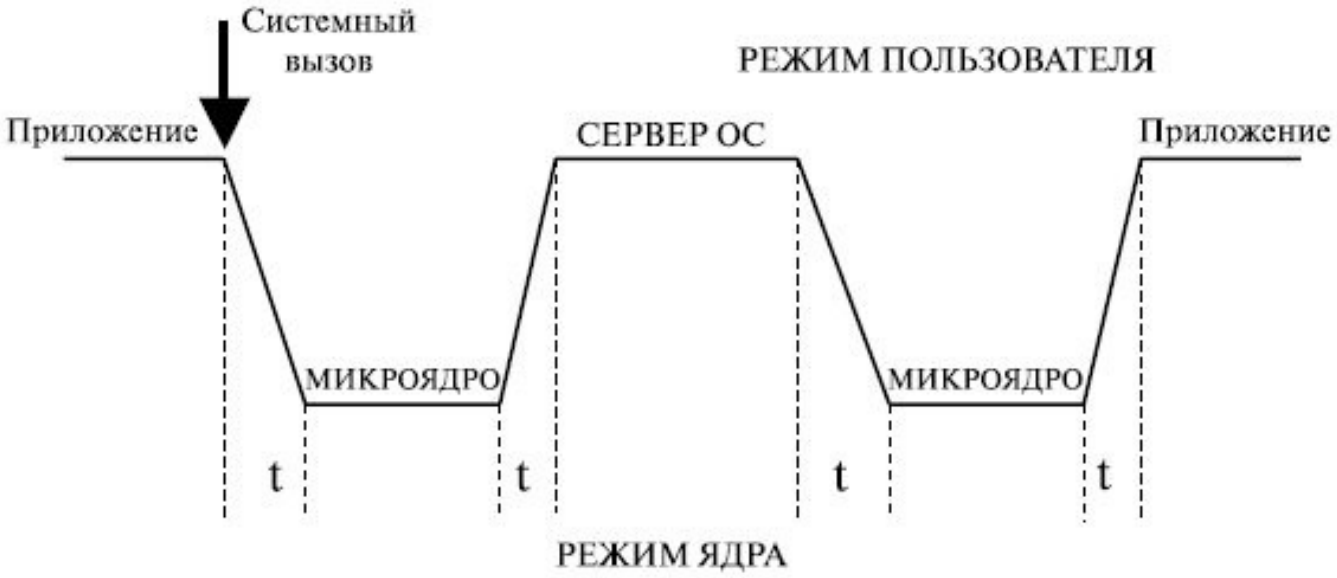

Рисунок 1.9 - Схема переключения режимов микроядерной архитектуры ОС

С 1990 года, в рамках проекта ***GNU***, ведется разработка микроядерная архитектура *Hurd ОС Linux*, основанная на микроядре *GNU Mach*. Ричард Столлман, руководитель проекта GNU, **в 2002 году** заявил о скором выходе производственной версии *Hurd*, однако его обещания не оправдались.

Микроядерными являются ядра *ОС Minix* и ядро *систем семейства BSD*.

Windows NT часто также называют микроядерной ОС, однако многие считают, что:

- *микроядро NT слишком велико* (более 1 Мбайт), чтобы носить приставку "микро";
- *все компоненты ядра работают в одном адресном пространстве* и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром.

1.6 Ядро и модули ОС

Как было отмечено выше, модули ядра выполняют основные базовые функции ОС:

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

Хотя модульная организация ОС характерна для микроядерной архитектуры, монолитные ядра также используют модули. Вызвано это тем, что аппаратная часть ЭВМ настолько многообразна, что практически неэффективно создавать ядро на все возможные варианты.

Традиционно, взаимодействие ядра ОС с аппаратной частью ЭВМ осуществляется через специальное ПО, которое называется **драйверами**.

Любое ядро ОС является одной большой программной, 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***.

Когда ядро ОС загружено в память ЭВМ и начинает работать, запускается первый пользовательский процесс ***init*** (или скрипт ***init***), имеющий PID=1, который обеспечивает дальнейшую загрузку необходимых модулей.

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.

1.7 Три базовых концепции ОС: файл, пользователь, процесс

Подведем итог изученного ранее учебного материала. Нами были расмотрены различные концепции и представления:

- *модель СОД* раскрывает различные архитектурные возможности применения средств вычислительной техники; на уровне отдельной ЭВМ можно выделить аппаратную и программную части; в программной части выделяется системное ПО, которое содержит объект нашего изучения — ОС;
- *концепции серверных ОС и ОС рабочих станций* в большей степени отражают особенности применения прикладного ПО, что находит свое отражение в дистрибутивах ОС и их использования в пределах СОД;
- *концепция многослойной структуры ОС* отражает современную парадигму построения сложных программных систем; стремление повысить надежность работы системного ПО приводит к выделению *ядра ОС*, работающего в привелигированном режиме; стремление повысить эффективность работы системного ПО приводит к созданию различных моделей ядер ОС, среди которых были выделены: *монолитное ядро ОС* и *микроядро ОС*;
- *концепция базового ПО ЭВМ* является попыткой формализовать и стандартизировать наиболее важные понятия ОС, к которым в первую очередь относится ее ядро; формализуется *интерфейс прикладного программирования* (API) и *идея абстрактной (виртуальной) машины*;
- *модели взаимодействия режимов ядра и пользователя* наглядно показывают архитектурные возможности построения ОС; демонстрируются архитектуры *классического (монолитного) ядра ОС* и *альтернативного — микроядра ОС*;
- *идея модульности ОС* является техническим решением, обеспечивающим устранение основного недостатка монолитных ядер — привязка к аппаратной конфигурации конкретной ЭВМ; в частности, *ядро ОС УПК АСУ является монолитным и модульным*.

Таким образом, нами изучены основные архитектурные особенности ОС, *кроме представлений прикладного уровня*, которые характерны для ПО, работающего в режиме пользователя. Данный подраздел посвящен именно этому вопросу.

Современный пользователь воспринимает ОС с внешней стороны — на уровне графической оболочки ОС. Эта оболочка представлена в виде *рабочего стола — **Desktop***, на котором имеются: окна, панели, меню, курсор мыши...
Некоторые ОС, например, *MS Windows и MacOS*, не могут запуститься без графической оболочки. В других, например, *UNIX и Linux*, графическая оболочка является пользовательским приложением: *X Window (X-сервер)*.

Если рассматирвать работу ПО ЭВМ на профессиональном уровне, то можно утверждать:

- *в привигелированном* режиме процессора (*режим ядра*) работает ядро ОС;
- *в не привигелированном* режиме процессора (*режим пользователя*) работают утилиты, инструментальное и прикладное ПО ЭВМ;
- *для прикладного ПО* — режим пользователя представляет некоторую *среду*

исполнения ОС, которой ядро ОС управляет;

- *прикладное ПО реализует некоторую модель СОД* и обращается к ядру ОС за дополнительными функциями, в основном связанными с доступом к аппаратным средствам ЭВМ;
- *обращение прикладного ПО к ядру ОС* осуществляется посредством *системных вызовов*;
- *классификация и абстрагирование* системных вызовов функций ядра ОС на более высоком уровне приводит к трем базовыми концепциям ОС: *файл, пользователь и процесс*.

Таким образом, мы пришли к понятию *среды исполнения ОС*, которая опирается на три базовых концепции файл, пользователь и процесс, образующие иерархию отношений, показанную на рисунке 1.10.

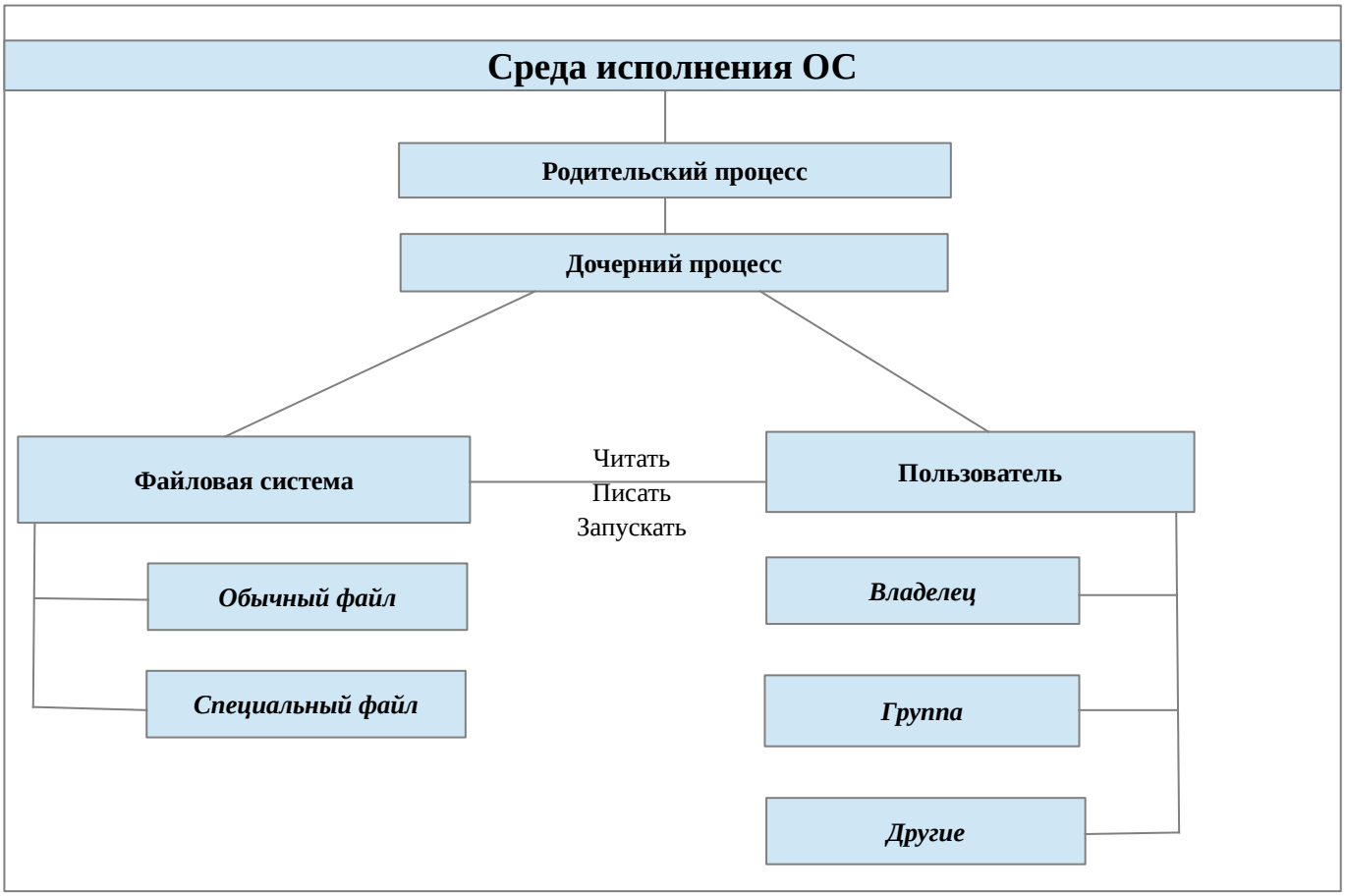

Рисунок 1.10 - Базовые концепции пользовательского режима ОС

Все концепции, структурно представленные на рисунке 1.10, будут подробно изучаться в последующих темах. Здесь мы рассмотрим лишь основные понятия и идеи, которые уже известны из теории других дисциплин или без которых невозможно обойтись при выполнении лабораторных работ.

Первое понятие, которое следует обсудить, яаляется концепция файла. ОС UNIX (Linux) прямо декларируют парадигму: «*Все есть файл*». К любому 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.

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.

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 про-

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

1.8 Системные вызовы fork(...) и exec*(…)

Ядро ОС самостоятельно запускает только один процесс ***init***. Все остальные *процессы режима пользователя* являются дочерними относительнопроцесса ***init***.

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

Вызов функции *fork(...)* из программы на языке С, имеет вид:

```
    #include <unistd.h>
    pid_t fork(void);
```

Функция *fork(...)* полностью дублирует существующий процесс, вместе со всеми открытыми файлами, порождая новый (дочерний) процесс с новым PID. Программист различает *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.

Замечание

Родительский процесс обязан дождаться завершения дочернего процесса, иначе контроль передается по иерархии выше. Процесс ***init*** является родителем для всех остальных процессов.

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

```
#include <unistd.h>
extern char **environ;

int execl(const char   *path, const char *arg, ...);
int execp(const char   *file, const char *arg, ...);
int execle(const char  *path, const char *arg, …, char * const envp[]);
int execv(const char   *path, const *char argv[]);
int execvp(const char  *file, const *char argv[]);
int execvpe(const char *file, const *char argv[], char * const envp[]);
```

Хорошо видно, что любой вызов имеет ссылку на файл, в качестве аргумента. После всех проверок на права запуска, указанный файл загружается в пространство дочернего процесса. Загруженной программе передаются все ресурсы дочернего процесса, включая открытые и созданные файлы.

Замечание

Обратите внимание, что родительский процесс продолжает контролировать работу уже новой программы и *имеет право принудительно завершить ее работу*.

1.9 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.

Учет всех перечисленных факторов может оказаться довольно сложной задачей и выходит за рамки нашей дисциплины. Более того, каждому дистрибутиву и его версиям посвящены отдельные сайты, а многие дистрибутивы постоянно обсуждаются на форумах в Интернете.

Для целей обучения выбран 64-битный базовый дистрибутив Arch Linux. На его основе создан набор ПО, организованный как учебный программный комплекс кафедры АСУ (УПК АСУ).

Структура УПК АСУ ориентирована не только на задачи курса «Операционные системы», но имеет все необходимое ПО для его изучения и организации лабораторных работ.

1.10 Лабораторная работа по теме №1

Данный учебный материал является методическим пособием по проведению лабораторной работы №1 по дисциплине «Операционные системы».
Работа проводится в рамках темы №1 «*Назначение и функции ОС*».
Цель работы — получение практических навыков использования ОС УПК АСУ, применительно к изучаемой дисциплине.
Указанная цель достигается посредством:

- *изучения структуры* ПО ОС УПК АСУ;
- *создания индивидуального загрузочного устройства* flashUSB;
- *освоения процедуры* запуска ОС УПК АСУ с загрузочного flashUSB;
- *получения навыков работы* в среде пользователя ***asu***; в частности: подключение к ОС УПК АСУ личного архива студента, содержащего рабочую область пользователя ***upk***; выход из сесси пользователя ***asu*** и вход в сессию пользователя ***upk***;
- *изучения рабочей среды* (рабочего стола) пользователя ***upk***, содержащего учебный материал и инструментальные средства для выполнения всех лабораторных работ по изучемой дисциплине;
- *выполнения учебных заданий* данного раздела пособия;
- *оформления отчета* по выполненным заданиям;
- *освоением процедур* создания личного архива на flashUSB и завершения работы с ОС УПК АСУ.

Замечание

Данное учебно-методическое пособие становится доступным только после запуска ОС УПК АСУ, подключения личного архива студента и входа в сеанс пользователя ***upk***, поэтому:

- значительная часть работ выполняется под непосредственным руководством преподавателя;
- основной учебный материал по данной лабораторной работе содержится в методическом пособии [3, раздел 1, «Назначение и использование ОС УПК АСУ»];
- учебный материал данного раздела только дополняет и уточняет [3], в плане особенностей изучаемой дисциплины.

Учитывая указанные выше ограничения, лабораторная работа №1 выполняетмя в три этапа.

Этап 1, студент:

- *передает* преподавателю личное устройство flashUSB для установки на него загрузочного ПО;
- *изучает* учебный материал первого раздела пособия [3], доступный на ЭВМ учебных классов кафедры АСУ как файл: *upk_asu.pdf*.

Этап 2, студент:

- *получает* от преподавателя личное устройство flashUSB с установленным на

нем загрузочным ПО и выполняет загрузку ОС УПК АСУ;

- *выполняет* учебные задания первого раздела пособия [3], доступного на рабочем столе пользователя ***asu*** как файл: *upk_asu.pdf*.

Этап 3, студент:

- *подключает* к ОС УПК АСУ личный архив со своего устройства flashUSB, используя учебный материал [3, подраздел 1.1]; архив должен находиться в корне файловой системы личного flashUSB: */asu64upk/themes/os-home.ext4fs*; при подключении архива будет запрошен пароль пользователя **asu**; следует ввести **upkasu**;
- *выходит* из сеанса пользователя ***asu*** и *входит* в сеанс пользователя ***upk***;
- *запускает* на чтение данное руководство, читает раздел 1 и выполняет задания лабораторной работы №1.

Замечание

В процессе выполнения лабораторных работ, студент использует сеансы пользователей *asu* и *upk*. Оба пользователя имеют пароль: ***upkasu***

1.10.1 Рабочий стол УПК АСУ

Рабочий стол пользователя ***upk*** для дисциплины «Операционные системы» показан на рисунке 1.11. Он имеет оригинальную для данного курса обучения заставку с надписью в верхней части экрана: «*Операционные системы. Тема os*». Наличие такого стилизованного фона говорит о правильном подключении рабочей среды пользователя ***upk*** и служит для визуального контроля выбора нужной учебной среды ОС УПК АСУ.

Замечание

В случае, когда архив рабочей среды пользователя ***upk*** создавался на ЭВМ с другим графическим адаптером, вместо указанной заставки может появиться изображение рабочего стола пользователя ***asu***, которое в ОС УПК АСУ установлено по умолчанию.
Следует восстановить нужное изображение и пересоздать личный архив.
Как это сделать? Обратитесь к преподавателю!

Кроме стилизованного изображения, на рабочем столе находится множество значков, часть из которых обозначают файловые системы компьютера. Другие, например, «*Домашний каталог*», «*Корзина*» и «*Файловая система*» находятся на рабочем столе всегда, а значек устройства flashUSB появляется только после подключения этого устройства. Нас, в первую очередь, должны интересовать *специальные значки* для данной дисциплины, перечень которых представлен в таблице 1.1.

Рисунок 1.11 — Правильное изображение рабочего стола

Таблица 1.1 — Специальные значки рабочего стола ОС УПК АСУ

Название значка	***Назначение***
Начало работы	Значек для запуска данного учебного пособия.
selfworks.pdf	Учебное пособие для самостоятельной и индивидуальной работы студента.
upk_asu.pdf	Учебное пособие [3].
Отчет.doc	Шаблон единого отчета для данной дисциплины, который студент должен регулярно и самостоятельно заполнять.
Учебный материал	Значек ссылки на директорию, содержащую весь учебный материал по данной дисциплине. Его активация запускает файловый менеджер Thunar.

Учебное задание

Запустить на редактирование файл *Отчет.doc*, а на просмотр «*Начало работы*» и *upk_asu.pdf*. Зафиксировать в отчете выполение задания по [3, подраздел 1.1]. Перейти к изучению подраздела 1.10.2 данного методического пособия.

1.10.2 Работа с личным архивом студента на flashUSB

Личный архив студента — файл с именем *os-home.ext4fs*, имеющий специальный формат хранения файловой системы ОС Linux, типа *ext4*.
Место хранения архива — личный flashUSB студента.
Каталог хранения архива — директория */asu64upk/themes* в файловой системе flashUSB, типа *FAT32*.

Студент запускает ОС УПК АСУ, используя ПО GRUB, как это описано в учебном руководстве [3].
ПО GRUB передает ядру ОС *набор параметров*, включая *UUID раздела* блочного устройства flashUSB, имеющего тип файловой системы *FAT32* и отмеченного как архивное устройство.
ОС УПК АСУ читает параметры ядра, в процессе запуска ОС, и сохраняет их для дальнейшего использования в файле */etc/upkasu/upkasu.conf*.
В частности, ОС запоминает *UUID раздела архивного блочного устройства*, который и использует для последующего поиска и подключения архива студента.

После нормального запуска, ОС УПК АСУ *автоматически подключает к системе* (login) пользователя с именем *asu*, как это описано в руководстве [3].
На рабочем столе пользователя остображается множество значков, среди которых имеется стилизованный значек личного flashUSB студента. Наведя курсор мыши на значек блочного устройства, можно узнать его состояние: подключено или не подключено.
Блочное устройство (съемный том) может находиться в двух состояниях:

- *подключено* — блочное устройство *подмонтировано* к некоторой директории файловой системы ОС и его содержимое *доступно* ПО ОС;
- *не подключено* — блочное устройство *не подмонтировано* к файловой системе ОС и его содержимое *не доступно* ПО ОС;

Подключение блочного устройства осуществляется левой кнопкой мыши, при этом запускается файловый менеджер ***Thunar***.
Меню работы с блочным устройством активируется правой кнопкой мыши, в котором имеются пункты:

- *Открыть* — при необходимости подключает устройство и запускает файловый менеджер Thunar;
- *Подключить том/Отключить том* — появляются в зависимости от состояния устройства, причем *подключение тома не запускает Thunar*;
- *Извлеч…* - появляется только для съемных устройств типа flashUSB;
- *Свойства* — доступно только для подключенных устройств;
- *Меню приложений* — второй уровень меню.

Замечание
Значек устройства «корневой» в файловой системы, не отображается на рабочем столе пользователя, поэтому flashUSB не отображается при аварийном варианте запуска.

Рабочая среда пользователя *asu* находится в памяти ЭВМ и расходует ее в процессе осуществления всех действий студента. Поэтому она используется только для выполнения следующих служебных операций:

- *подключение/отключение* flashUSB по отношению к ЭВМ;
- *подключение* архива темы обучения к рабочей среде пользователя ***upk***;
- *отключение* архива темы обучения от рабочей среды пользователя ***upk***;
- *выключение* компьютера;
- *проведение служебных операций*: форматирование flashUSB, создание файловой системы на flashUSB, установка на flashUSB ПО GRUB и ПО аварийного варианта загрузки ОС.

При запуске ОС УПК АСУ, рабочая среда пользователя *asu* восстанавливается из архива ОС.

Рабочая среда пользователя *upk* присутствует и используется во множестве вариантах:

- *базовый вариант* - рабочая среда пользователя ***upk***, которая восстанавливается из архива ОС, во время ее запуска;
- *учебные варианты* — рабочие среды, находящиеся в личных архивах студентов, которые подключаются и отключаются от системы из среды пользователя ***asu***.

Назначение базового варианта рабочей среды — предупреждение студента, что он или забыл подключить тему обучения, или — при подключении темы возникли проблемы.

Назначение учебных вариантов рабочих сред — подключение их к ОС УПК АСУ с целью:

- *обеспечение студента* учебным материалом и инструментами, во время проведения занятий по конкретной дисциплине;
- *оперативное сохранение* данных в личном архиве студента, во время проведения учебного занятия.

Замечание

Личный архив студента, размещенный на его личном flashUSB, имеет ограниченный объем файловой системы, большая часть которой занята учебным материалом и системным ПО ОС.

Первоначальный размер файла архива *os-home.ext4fs* - 300 Мбайт, поэтому:

- ***не следует хранить*** в личном архиве пользователя *upk* посторонние файлы, кроме тех, что предусмотеры для проведения занятий по изучаемой дисциплине;
- ***обязательно сохранять*** на личном flashUSB студента, за пределами архива, копию отчета по даной дисциплине, шаблон которого представлен на рабочем столе пользователя *upk*, в виде файла *Отчет.doc*;
- ***в случае*** повреждения архива или его полного заполнения, *следует обратиться к преподавателю* для консультаций по устранению возникших проблем.

Перед окончанием занятия, студент должен:

- *закрыть* все окна и остановить все приложения в среде пользователя *upk*;

- *выйти* из среды пользователя *upk* и зайти в среду пользователя *asu*;
- *отключить* личный архив, воспользовавшись значком на рабочем столе;
- *отключить* личный flashUSB, воспользовавшись значком его блочного устройства, также расположенном на рабочем столе;
- *выключить* компьютер.

1.10.3 Изучение рабочей среды пользователя upk

Учебное задание

Находясь в середе пользователя ***upk***, выполнить задания подразделов 1.2 и 1.3 из учебно-методического пособия [3].
Прочитать и усвоить учебный материал раздела 3 из пособия [3].
Отразить результаты работы в своем личном отчете.

Для изучения рабочей среды пользователя имеется три основных инструмента:

- *командная строка* (виртуальный терминал, консоль), в которой можно запускать команды языка shell;
- *текстовый файловый менеджер* (Midnight Commander), который запускается в окне виртуального терминала (командной строке) командой **mc**;
- *графический файловый менеджер* (Thunar), который запускается разными способами, например, активацией левой кнопкой мыши значка «Домашний каталог», расположенного на рабочем столе любого пользователя.

Инструмент командной строки является самым главным в нашем курсе обучения, поскольку командные языки shell (*sh*, *bash* и другие) являются основой управления системным ПО любой ОС.
В частности, изучению языка ***sh*** посвящена «*Тема 3*» нашего курса обучения.
В качестве примера, рассмотрим один из способов текущего контроля *размера использованной области* пользователя *upk*:

```
du -hs /home/upk
```

Вывод этой команды показан на рисунке 1.12.

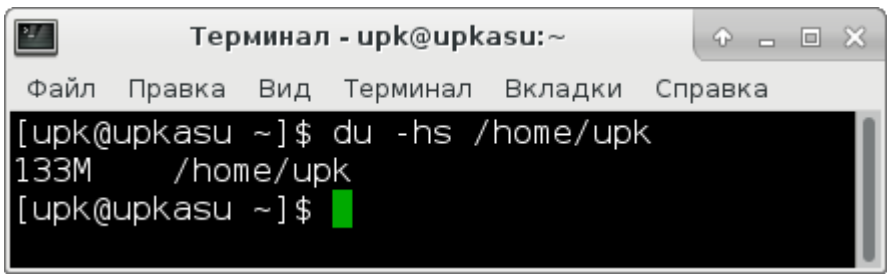

Рисунок 1.12 — Контроль размера содержимого отдельного каталога

Другой пример. На рисунке 1.13 показан вывод команды: ***`ls -la`***.

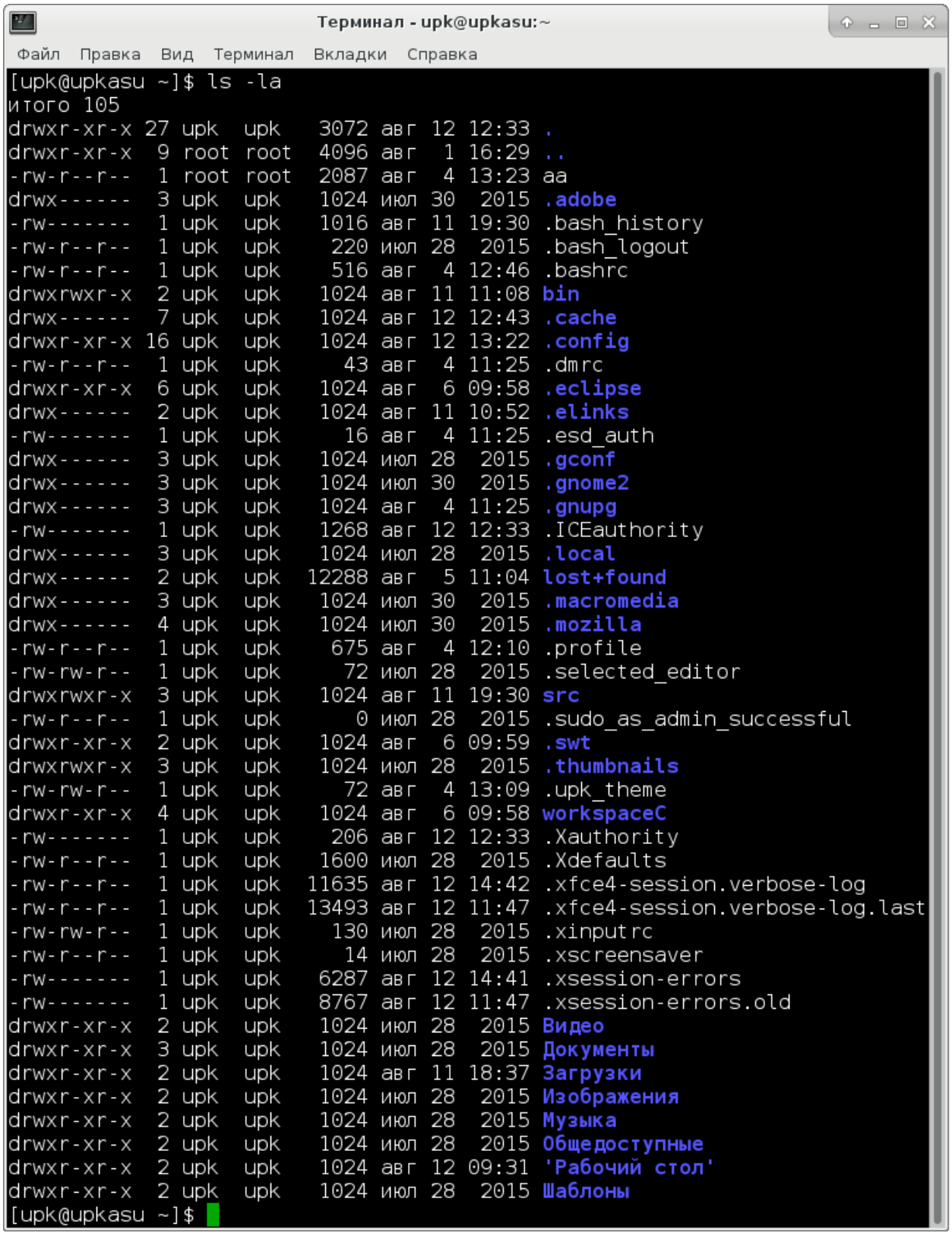

Рисунок 1.13 — Контроль содержимого домашнего каталога пользователя upk

Завершая первую лабораторную работу, рассмотрим основное место хранения учебного материала по данной дисциплине. Оно расположено в домашней директории пользователя *upk*: *~/Документы*.

Доступ к этой директории проще всего получить, активировав левой кнопкой мыши значек «*Учебный материал*», расположенный на рабочем столе пользователя.
В результате, запустится файловый менеджер ***Thunar***, который покажет содержимое директории */home/upk/Документы*, как показано на рисунке 1.14.

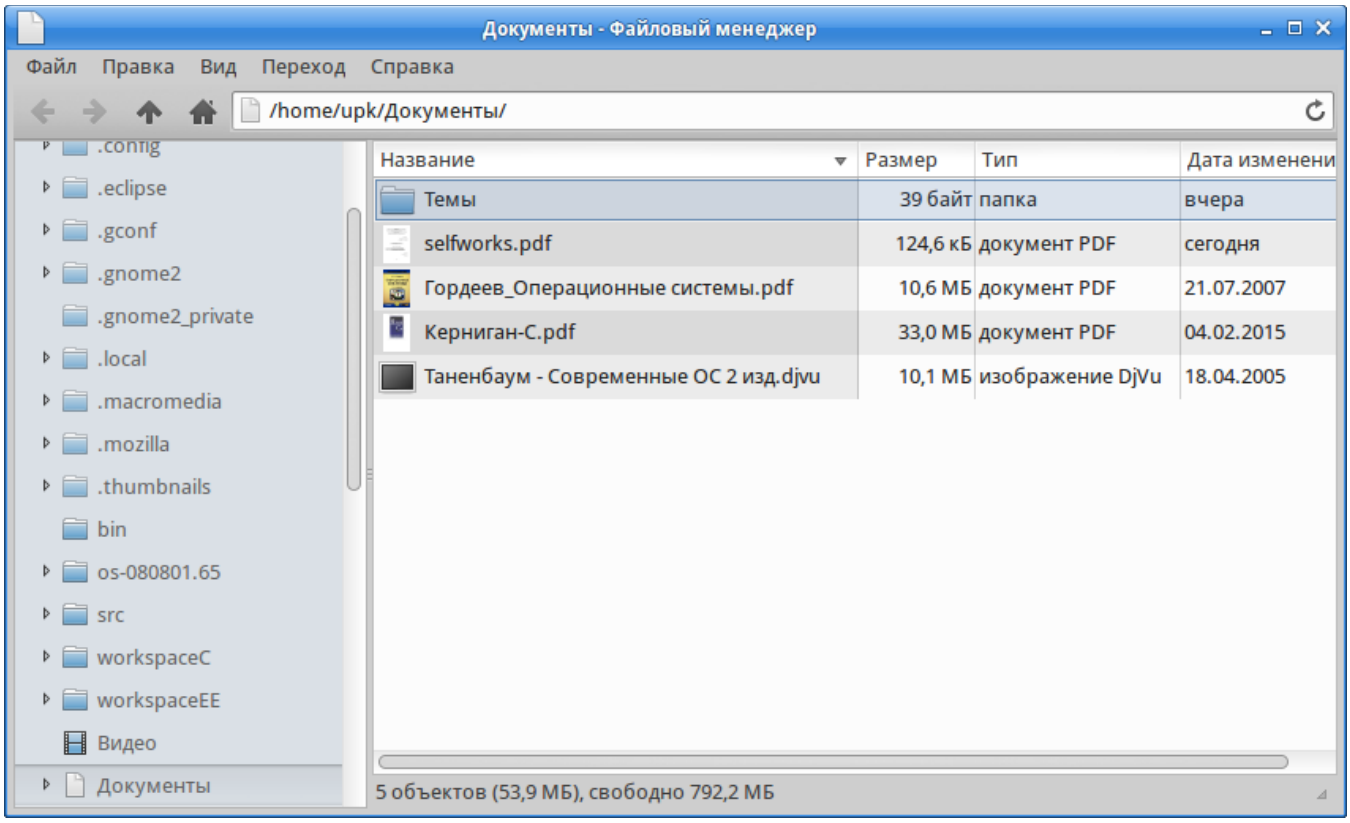

Рисунок 1.14 — Директория «Документы» пользователя upk

В этой директории находятся:

- *общая учебная литература* по данной дисциплине;
- *директория* «*Темы*», в которой находятся файлы учебно-методических пособий.

Обозначения файлов директории «Темы»:

`Тема`**`X`**`_os.pdf`,

где ***X*** — номер изучаемой темы по дисциплине «*Операционные системы*».

Далее, следует:

- отобразить в отчете изученный материал;
- выключить компьютер, завершив лабораторную работу №1.

2 Тема 2. BIOS, UEFI и загрузка ОС

В предыдущей теме было рассмотрено множество моделей касающихся как ОС, так и компьютера в целом. В частности, было утверждение, что *ОС является некоторой виртуальной машиной*, которая установлена на аппаратных средствах некоторой ЭВМ. Большинство современных ОС могут работать на аппаратных средствах ЭВМ различных архитектур, при этом, пользователь может и не знать об этих различиях, поскольку внешние проявления в поведении таких ОС могут быть неразличимы.
Достигается это специальной архитектурой ядра ОС, в которой:

- *стандартизируется интерфейс* (API) между режимами ядра и пользователя;
- *выделяется программная прослойка* между ядром ОС и аппаратным обеспечением ЭВМ, реализуемая с помощью *модулей (драйверов)*, которые или статически компилируются с ядром или загружаются по мере необходимости.

С другой стороны, прежде чем ОС начнет функционировать, ее ядро должно быть загружено в память ЭВМ. Для этого служит специальное ПО, которое входит в состав самого компьютера и является его неотъемлемой частью, независимо от установленной ОС.

В данной теме, мы рассмотрим такое ПО, которое появилось с развитием средств микропроцессорной техники, позволившей заменить специальные аппаратные средства загрузки ОС на программные.
По традиции, изложение учебного материала будет дано *для архитектуры процессора x86*, модификации которого установлены в компьютерных классах кафедры АСУ и широко используются в переностных ЭВМ.

2.1 Архитектура x86

Чтобы выйти на должный уровень понимания, следует рассмотреть ряд вопросов, касающихся аппаратной части компьютеров. Традиционно, в качестве основной модели компьютера, рассматриваются изделия созданные на базе процессора ***x86***.

x86 (***Intel 80x86***) — архитектура процессора с одноименным набором команд, которая впервые была реализованна в процессорах компании *Intel* и относится к серии ее процессоров ранних моделей — ***80086, 80386 (i386), 80486 (i486)***.

На рисунке 2.1, показаны *16-битные регистры* процессора ***80086***, обеспечивающие работу компьютера на *20-битной шине адреса*:

- *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.

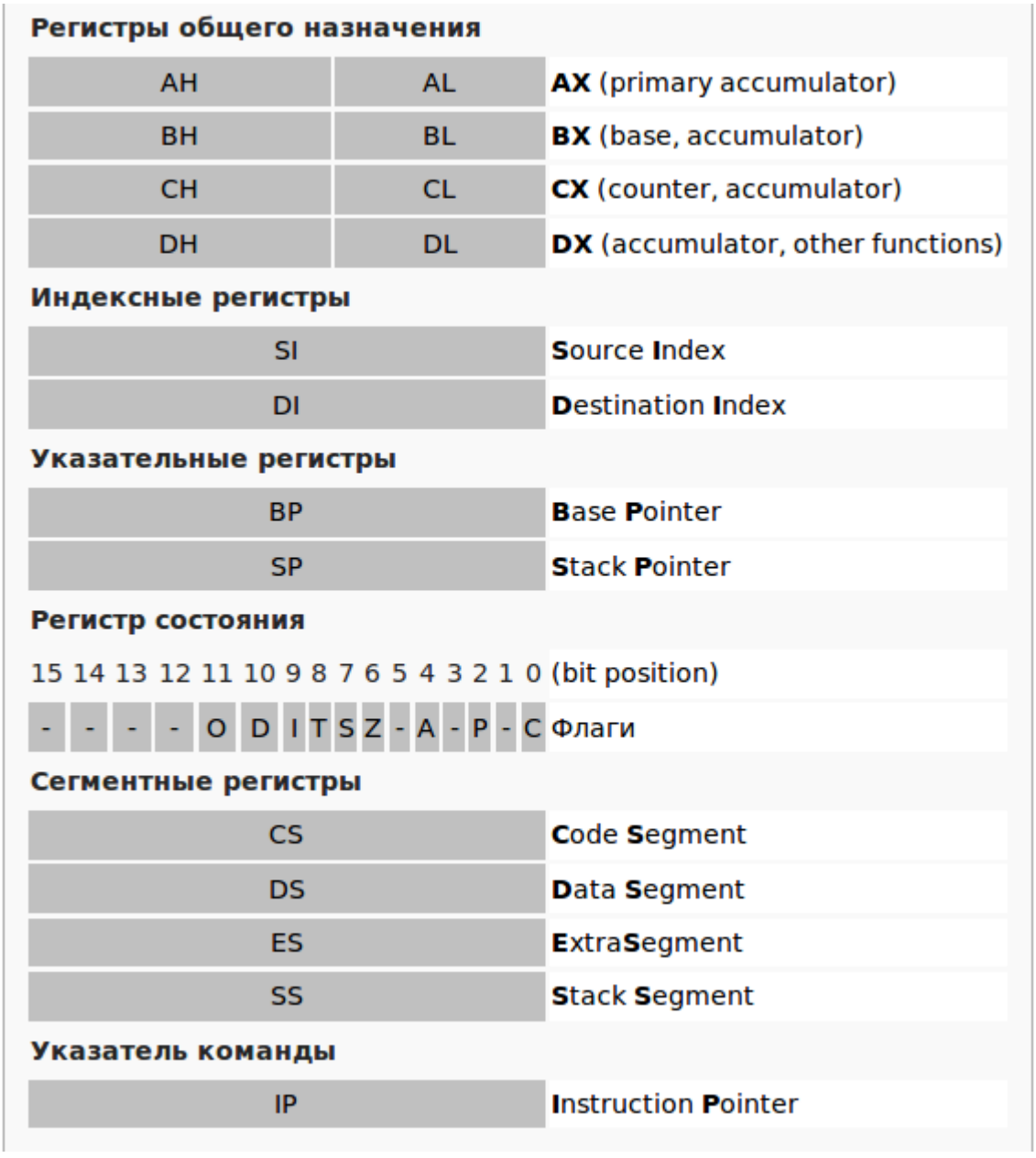

Рисунок 2.1 - Регистры процессора 80086

В общем случае, для работы с регистрами процессора необходимо знать набор его команд, которые на практике пишутся на языке ассемблера.
Поскольку изучение ассемблера не входит в программу нашего обучения, то мы отметим основные качественные характеристики этого процессора:

- объем адресуемой памяти *1 Мбайт*;
- работа в реальном режиме: *защищенный режим работы отсутствует*.

Преемником компании ***Intel***, для дешевых процессоров x86, стала компания ***AMD***. *Со временем*, появились ***32-битные*** и ***64-битные*** процессора.
Начиная с процессора 80386, появился полноценный защищенный режим работы, который стал обеспечивать ядрам ОС *привелигированный режим работы*.

Для 64-битных процессоров и соответствующего ПО, совместимых с набором команд ***x86***, стало применяться обозначение ***x86-64***. На рисунке 2.2 (а), показаны регистры процессора ***AMD64***.

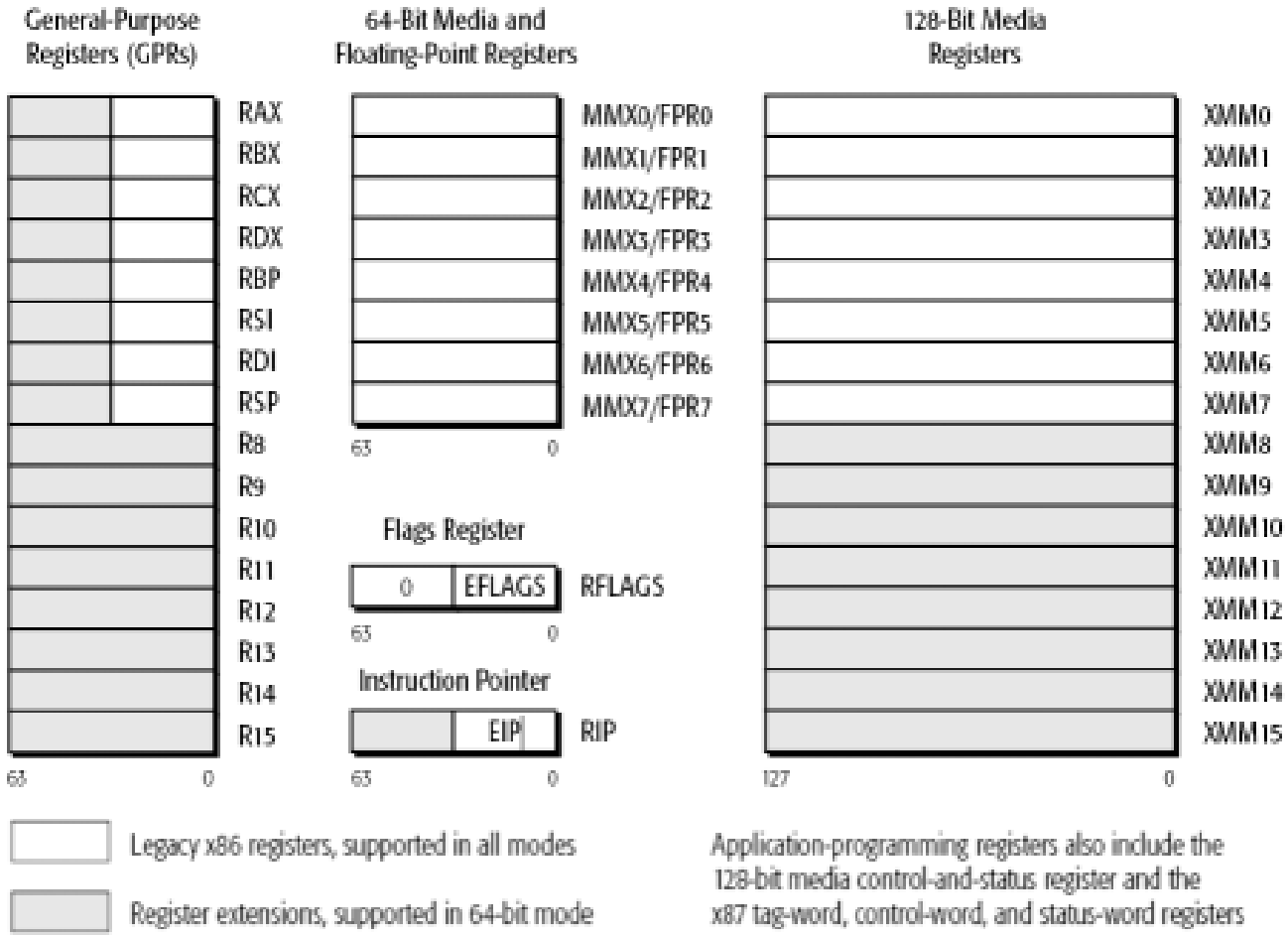

Рисунок 2.2 (а) - Набор регистров процессоров x86-64

Замечание

Хотя 32-битное ПО может работать на процессорах x86-64, *64-битное ПО не может работать на процессорах x86*. Несмотря на расширение количества регистров и увеличение их размеров, имеется подмножество команд, которые выполняются на всех процессорах. Такой набор команд соответствует *процессору 80386*, а набор команд часто в дистрибутивах ОС обозначается как *i386*.

На рисунке 2.2 (б) показан полный набор регистров процессовов x86.

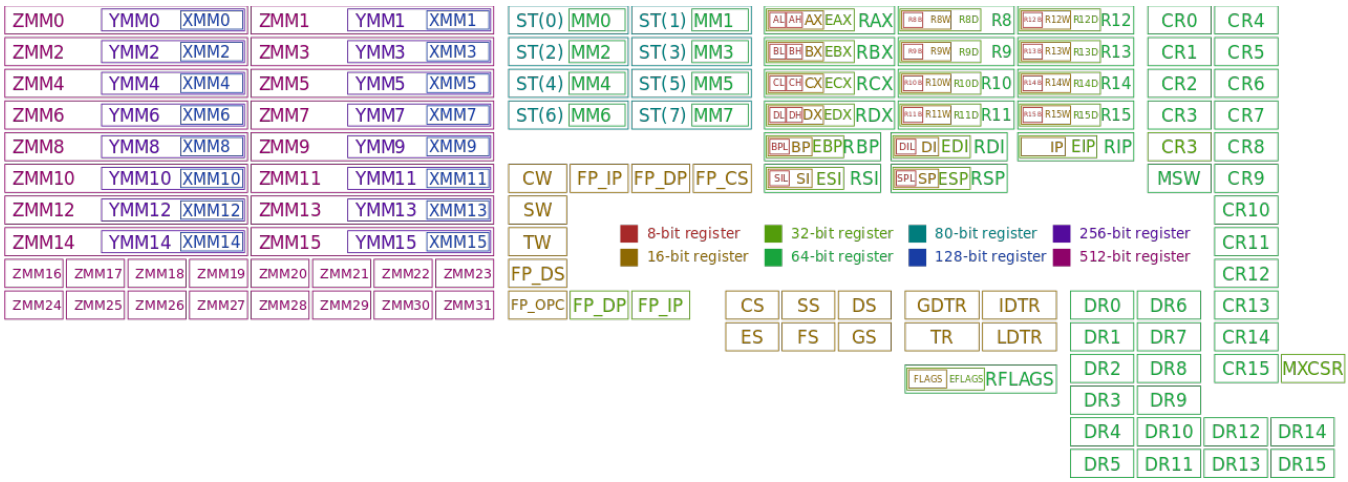

Рисунок 2.2 (б) — Полный набор регистров процессоров x86

Среди всего множества команд, которые типично работают на всех процессорах и используют сегмент стека, *выделяются две*:

- *вызов функции* или команда: **CALL адрес**; в стек заносятся сегмент и смещение следующей за вызовом функции команды, а затем управление передается по адресу указанному в команде CALL; когда функция заканчивает работу, то из стека извлекаются сохраненные сегмент и смещение, которые помещаются в регистры CS и IP, обеспечивая продолжение работы программы;
- *прерывание* или команда: **INTERRUPT номер**; кроме сегмента и смещения команд, в стек заносятся регистр флагов, тем самым сохраняя полное состояние процессора; **номер** соответствует *номеру слова* в начале памяти ЭВМ, в которые предварительно записаны значения сегмента и смещения, соответствующие адресу обработчика прерывания; таким образом обычно обрабатываются аппаратные прерывания процессора.

Другим важным аспектом работы аппаратного обеспечения ЭВМ является *способ выполения операций ввода-вывода*.
Существует три таких способа:
Программируемый ввод-вывод, когда процессор посылает команды, связанные с ним контроллеру, а затем *периодически проверяет состояние модуля ввода-вывода с целью проверки завершения операции*.

Ввод-вывод, управляемый прерываниями, когда процессор посылает необходимые команды контроллеру ввода-вывода и продолжает выполнять текущий процесс, если нет необходимости в ожидании выполнения операции ввода-вывода. В ином случае, текущий процесс приостанавливается до получения сигнала прерывания о завершении ввода-вывода, а процессор переключается на выполнение другого процесса. Наличие прерываний процессор проверяет в конце каждого цикла выполняемых команд.

Прямой доступ к памяти (direct memory access – DMA). В этом случае, имеется специальный *аппаратный модуль прямого доступа к памяти*, который управляет обменом данных между основной памятью и контроллером ввода-вывода. При

этом, процессор посылает запрос на передачу блока данных модулю DMA, а само прерывание происходит только после передачи всего блока данных.

В современных компьютерах используется прямой способ доступа к памяти, схема которого показана на рисунке 2.3.

Подсистема ввода-вывода должна также учитывать *режим работы шины*, который может быть *пословным* или *поблочном*.
В пословном режиме, контроллер DMA выставляет запрос на перенос одного слова и получает его. Если процессору также нужна эта шина, ему приходится подождать.
Такой механизм называется ***захватом цикла***, потому, что контроллер устройства периодически забирает случайный цикл шины у центрального процессора, слегка тормозя его. На рисунке 2.4, показана позиция цикла команд, в которых работа процессора может быть приостановлена.
В любом случае, приостановка процессора происходит только при необходимости использования шины. После этого, устройство DMA выполняет передачу слова и возвращает управление процессору.

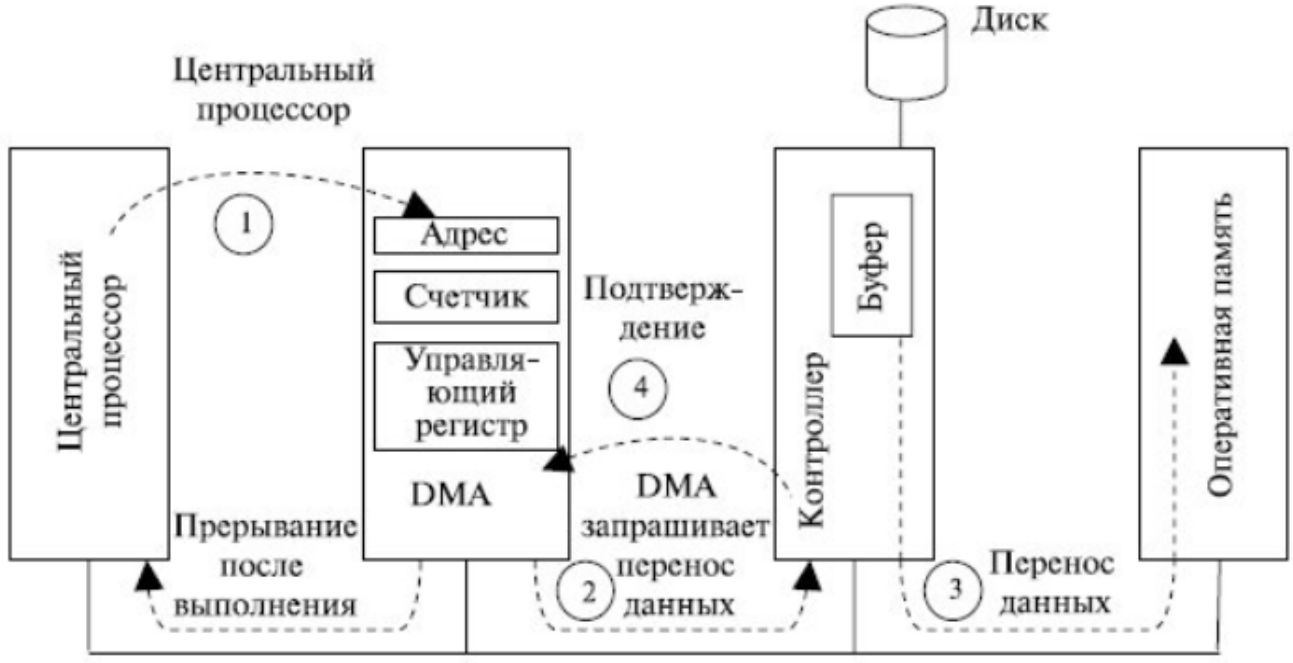

Рисунок 2.3 - Схема прямого доступа к памяти с помошью DMA

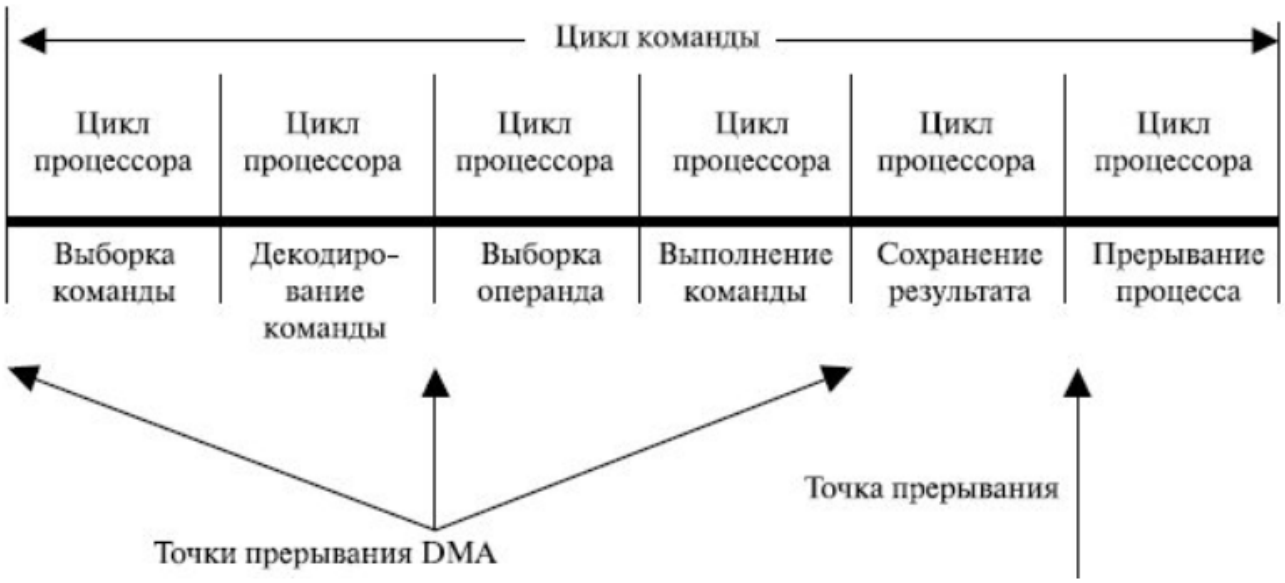

Рисунок 2.4 - Точки прерывания процессора устройством DMA

В блочном режиме, работы контроллер DMA занимает шину на серию пересылок (***пакет***). Такой режим более эффективен, однако при переносе большого блока данных центральный процессор и другие устройства могут быть заблокированы на существенный промежуток времени.

При большом количестве устройств, от подсистемы ввода-вывода *требуется спланировать свою работу в реальном масштабе времени*, в котором работают все внешние устройства, запуск и приостановку большего количества разных драйверов, обеспечив, при этом, время реакции каждого драйвера на независимые события контролеров внешних устройств. С другой стороны, необходимо минимизировать загрузку процессора задачами ввода-вывода. Решение этих задач достигается на основе *многоуровневой приоритетной схемы обслуживания прерываний*: для обеспечения приемлемого уровня реакции, все драйверы распределяются по нескольким приоритетным уровням, в соответствии с требованиями по времени реакции и временем использования процессора.
Для реализации приоритетной схемы управления используется ***общий диспетчер прерываний ОС***.

Замечание

Более подробное рассмотрение алгоритмов взаимодействия подсистемы ввода-вывода с процессором, шиной и контроллерами внешних устройств требует знания и использования *диаграмм сигналов*, а также знания и практику работы с языком ассемблера. Все эти вопросы выходят за рамки нашего курса.

2.2 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.

Имеется два типа перезаписываемых микросхем, хранящих BIOS:

- *микросхемы EPROM (Erasable Programmable Read Only Memory)*: содержимое этих микросхем может стерто при помощи ультрафиолетового излучения специальным прибором (*старый вариант*);
- *микросхемы EEPROM (Electrically EPROM)*: содержимое этих микросхем может быть стерто при помощи электрического сигнала, при этом микросхему не обязательно вынимать из компьютера.

Когда появились первые персональные компьютеры, необходимость в BIOS стала критической. Производители компьютеров стали использовать продукты BIOS трех производителей: ***AMI***, ***AWARD*** *и* ***Phoenix***. 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.
CMOS - *complementary metal-oxide-semiconductor* - технология построения электронных схем или КМОП - *комплементарный металлооксидный полупроводник*. Кроме настроек BIOS в CMOS хранятся *параметры конфигурации компьютера*. Суммарный объем памяти CMOS составляет всего 256 байт и потребляет она очень мало энергии. Стандартная батарейка, расположенная на материнской плате питает CMOS в течение 5-6 лет, после чего необходимо производить ее замену.

Замечание

Если срок батарейки, питающей CMOS, подошел к концу, то при включении ЭВМ на экран будет выведено сообщение, например, "*CMOS-checksum error*".
Для возобновления работы компьютера необходимо будет установить новую батарейку взамен вышедшей из строя.

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. Справочную информацию по настройке можно найти в инструкции к материнской плате или в сети.

Программа настройка BIOS (*BIOS Setup*) может быть вызвана после перезагрузки компьютера нажатием определенной клавиши или группы клавиш.

Наиболее распространенные — **Del**, **F2** или **Esc**.

Существуют также определенные комбинации клавиш, позволяющие:

- *запустить микропрограмму восстановления* (перезаписи) BIOS в микросхеме в случае повреждения ее, аппаратно либо вирусом;
- *восстановить заводские настройки*, позволяющие запустить компьютер после неверных настроек.

Замечание

Неверные настройки BIOS могут нарушить работу компьютера.

2.3 Этапы и режимы POST

Основную часть BIOS материнской платы составляют микропрограммы инициализации контроллеров на материнской плате. Подключенные к материнской плате устройства, в свою очередь, могут иметь *управляющие контроллеры с собственными BIOS*.

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.

Замечание

Выбор между прохождением *полного* или *сокращенного* набора тестов, при включении компьютера, можно задать в программе настройки базовой системы ввода-вывода, *Setup BIOS*.

2.4 UEFI и его стандартизация

Для новых современных платформ ЭВМ, компания *Intel* предлагает **EFI** — *Extensible Firmware Interface*.
Первоначально, *середины 1990 годов*, EFI разрабатывалась для первых систем *Intel-HP Itanium*.
Позже, этот интерфейс был переименован в ***UEFI***, разработку которого продолжил *Unified EFI Forum*.
На данный момент, *последняя версия UEFI 2.4*, принята **в июле 2013 года**.

Обычно, UEFI имеет новый графический интерфейс, предполагающий *улучшить «реликтовый BIOS»*. Это стало возможным благодаря совершенствованию технологии изготовления микросхем EEPROM: увеличению их объема и быстродействия, а также снижению себестоимости. Тем не менее, между функционированием BIOS и UEFI имеются существенные различия, которые необходимо хорошо знать. Рассмотрим это подробнее.

Процессор x86, после включения питания ЭВМ, проводит самотестирование и начинает свою работу *в реальном режиме*, который обеспечивает ему доступ ко всем ресурсам компьютера. Обнулив все регистры, он выставляет значения CS и IP специальным образом:

- *для моделей до 80386-DX*: CS=0xFFFF, IP=0x0000 — что указывает на последние 16 байт в конце 1-го МБайта оперативной памяти ЭВМ;
- *начиная с 80386-DX*: CS=0x0000, EIP=0xFFFFFFF0 — что указывает на последние 16 байт в конце 4-х ГБайт оперативной памяти ЭВМ.

После установки начальных значений регистров и захвата шины компьютера, процессор начивает выполнять команды извлекаемые из памяти ЭВМ и эта работа не прекращается до полной остановки самого процессора.
Обычно, указанные 16 байт, содержат команду ***GOTO*** по адресу ПО BIOS, что поддерживается специальной микросхемой памяти, определенной аппаратным конструктивом компьютера.
Таким образом, начинается работа любой современной ЭВМ.

ПО BIOS, начиная свою работу, делает небольшой тайм-аут выводит на экран подсказку, чтобы пользователь мог войти в режим настройки (*BIOS Setup*). Выполнив все программы POST, BIOS ищет загрузочное устройство ЭВМ, среди списка доступных, после чего запускает загрузочный код, расположенный в специальном секторе блочного устройства: MBR.
MBR — *Master Boot Record* — специальная структура загрузочного устройства, подробно рассмотренная далее.

Возможности ПО BIOS достаточно широки и не ограничиваются только перечисленными выше функциями POST, поиском загрузочного устройства и запуском программного кода MBR. Чтобы это показать, рассмотрим структуру ОЗУ ЭВМ.
Типичная схема оперативной памяти (ОЗУ) IBM PC-совместимого компьютера показана на рисунке 2.5.
В начале ее расположена *область векоторов прерываний*, занимающая 1024 байта: по 4 байта на один вектор (всего 256 векторов).
Вектор прерывания — адрес программы в памяти ОЗУ (*обработчика прерывания*), которая будет исполняться процессором, когда такое прерывание произойдет.
BIOS в начале своей работы, выставляет адреса этих векторов на свое собственной ПО, обеспечивая возможности полнофункционального управления компьютером.

Для компьютеров типа IBM PC AT, назначение ряда векторов прерываний следующее:

- **INT 00h** — деление на 0;
- **INT 01h**— пошаговый режим;
- **INT 02h**— немаскируемое прерывание;
- **TXTT ATI INT O3h** — точка останова;
- **INT 04h** — переполнение;
- **INT 08h** — таймер;
- **INT 09h** — клавиатура;
- **INT 10h** — видео сервис;
- **INT 33h** — поддержка мыши;
- **INT 4Ah** — будильник пользователя.

Первоначально, такие ОС, как MSDOS, использовали эти прерывания в своей работе. MSDOS имеет даже свой собственный вектор ***21h*** — функции DOS.

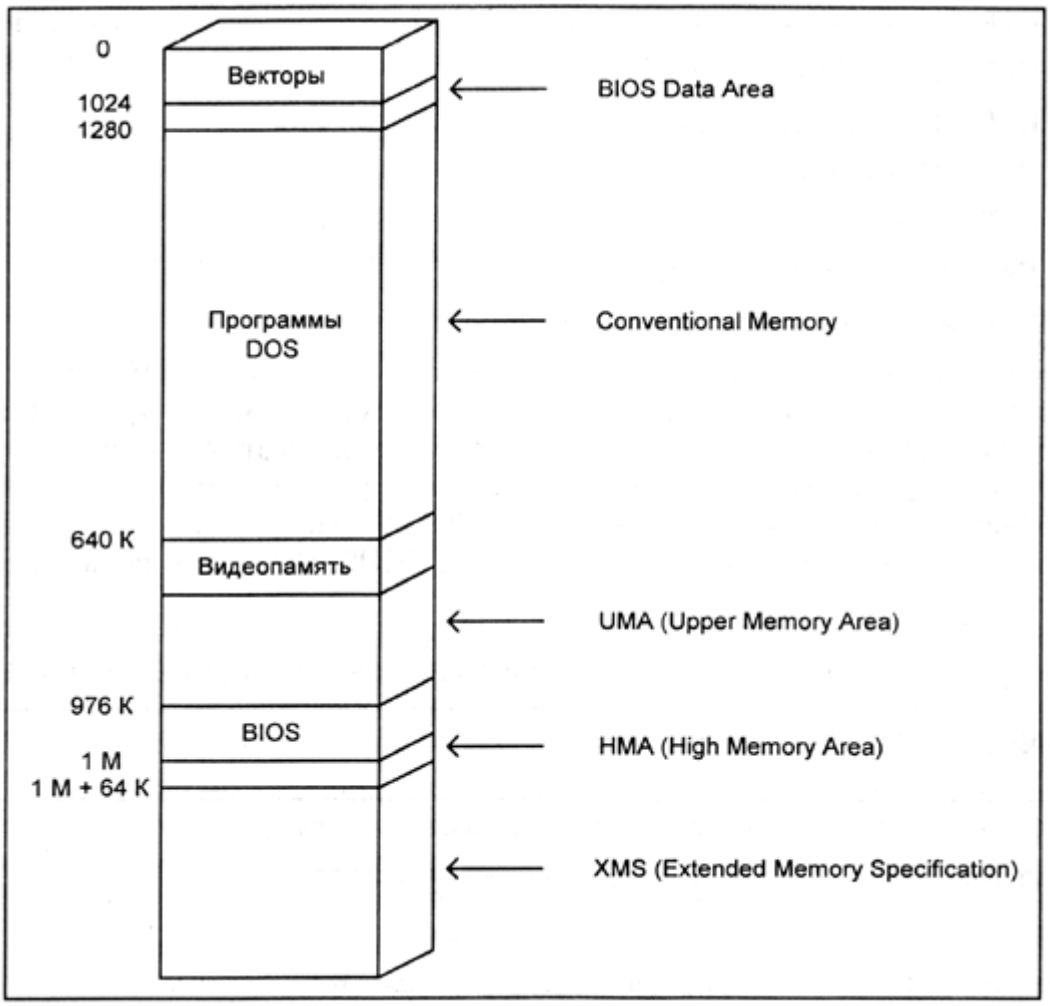

Рисунок 2.5 — Типичная схема ОЗУ памяти ЭВМ

ПО UEFI интенсивно использует новейшие технологические возможности современных компьютеров. Хотя также выполняются функции аналогичные POST, UEFI сразу переводит процессор в защищенный режим работы, тем самым, обеспечивая повышенную надежность работы ПО. Такой подход позволяет использовать сертифицированные подходы к использованию ЭВМ, буквально на этапе его включения. *Кроме того*, ПО UEFI способно работать с файловыми системами и более современной структурой блочных устройств — GPT.
GPT — *GUID Partition Table* — новая структура блочных устройств, позволяющая разбивать их на *128 основных разделов*, в отличие от структуры MBR, которая допускает наличие *только 4-х основных разделов*.

Можно выделить следующие основные особенности ПО UEFI:

- *работа в защищенном режиме* процессора;
- *возможность защищенной сертификатами* загрузки ОС;
- *возможность поддержки CSM* - *Compatibility Support Module* — модули, обеспечивающие загрузку ОС, совместимую с возможностями BIOS;
- *модульная организация ПО* поддержки аппаратных средств компьютера, *firmware*, которое можно собирать и устанавливать от производителей оборудования;

- *специальная структура ПО*, соответствующая запускаемым *exe-файлам* ОС MS Windows;
- *возможность загрузки множества* разных ОС;
- *непосредственный вызов* загрузчика ОС;
- *возможность работы* со структурой GPT блочных устройств;
- *требование наличия собственного специального раздела* для блочных устройств;
- *требование формата FAT12/16/32* для собственного раздела.

Замечание

1. Стали появляться ноутбуки с UEFI и предустановленной ОС MS Windows 8. При покупке такого компьютера, убедитесь, что основное окно UEFI имеет переключатель «*Legacy BIOS*» и отключение режима «*Security*», иначе вам не удастся загрузить другую ОС.
2. Для компьютеров на базе иных платформ, чем ***x86***, для обозначения встроенного ПО могут использоваться другие термины. Например, для архитектур *SPARC* или *UltraSPARC*, *firmware* может называться ***PROM*** или ***Boot***.

2.5 Блочные и символьные устройства компьютера

В предыдущей теме, перечисляя типы специальных файлов, мы отметили *файлы устройств*.
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 (винчестера)» имеют *физические блоки фиксированной длины*.
Текущий стандарт физического блока винчестера: *1 сектор — 512 байт*.
Символьное устройство *не обозначает текстового содержимого*.
Для симольных 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:, D:, 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 ***udev*** 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.

2.6 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.

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 (2^{48}) 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:

$$LBA(c,h,s) = (c \cdot H + h) \cdot S + s - 1$$
$$s = (LBA \bmod S) + 1$$
$$h = \frac{(LBA + 1 - s) \bmod (H \cdot S)}{S}$$
$$c = \frac{LBA + 1 - s - h \cdot S}{H \cdot S}$$

где:
c — 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*.

2.7 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 2.1):

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

Таблица 2.1 - Структура Главной загрузочной записи (MBR)

Адрес	***Содержимое***
0x0000	Код загрузчика
0x01B8	4-х байтная сигнатура диска (только для MS Windows 2000 и XP)
0x01BE	Четыре 16-байтных записи схемы таблицы основных разделов MBR **(partition table)**
0x01FE	2-х байтная сигнатура MBR ($55AA_{16}$)

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 редактиру-*

ется в момент инсталляции ОС на внешний носитель.

Когда BIOS прочитает первый сектор блочного устройства и запишет его по адресу 0000:7C00, 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 2.2: *если загрузочный раздел найден*, то первый сектор загрузчика ОС записывается по адресу 0000:7C00 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 2.3.
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.

Таблица 2.2 - Структура описания раздела

Смещение	***Длина***	***Описание***
00h	1	***Признак активности*** раздела
01h	1	Начало раздела — головка
02h	1	Начало раздела — сектор (биты 0-5), дорожка (биты 6,7)
03h	1	Начало раздела — дорожка (старшие биты 8,9 хранятся в байте номера сектора)
04h	1	***Код типа раздела*** – код файловой системы
05h	1	Конец раздела — головка
06h	1	Конец раздела — сектор (биты 0-5), дорожка (биты 6,7)
07h	1	Конец раздела — дорожка (старшие биты 8,9 хранятся в байте номера сектора)
08h	4	Смещение первого сектора
0Ch	4	Количество секторов раздела

Таблица 2.3 - Ранее распространенные коды типов файловых систем

ID (hex)	***Описание***
01	Primary DOS12 (12-bit FAT)
04	Primary DOS16 (16-bit FAT)
05	Extended DOS
06	Primary big DOS (> 32MB)
0A	OS/2®
83	Linux (EXT2FS)
A5	FreeBSD, NetBSD, 386BSD (UFS)

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 2.4.

Таблица 2.4 - Структура EBR

Смещение	***Длина***	***Описание***
1BEh	16	Указатель на раздел
1CEh	16	Указатель на следующий EBR
1FEh	2	Сигнатура (55h AAh)

Замечание

Формат указателей в таблице 2.4 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 2.6.
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.

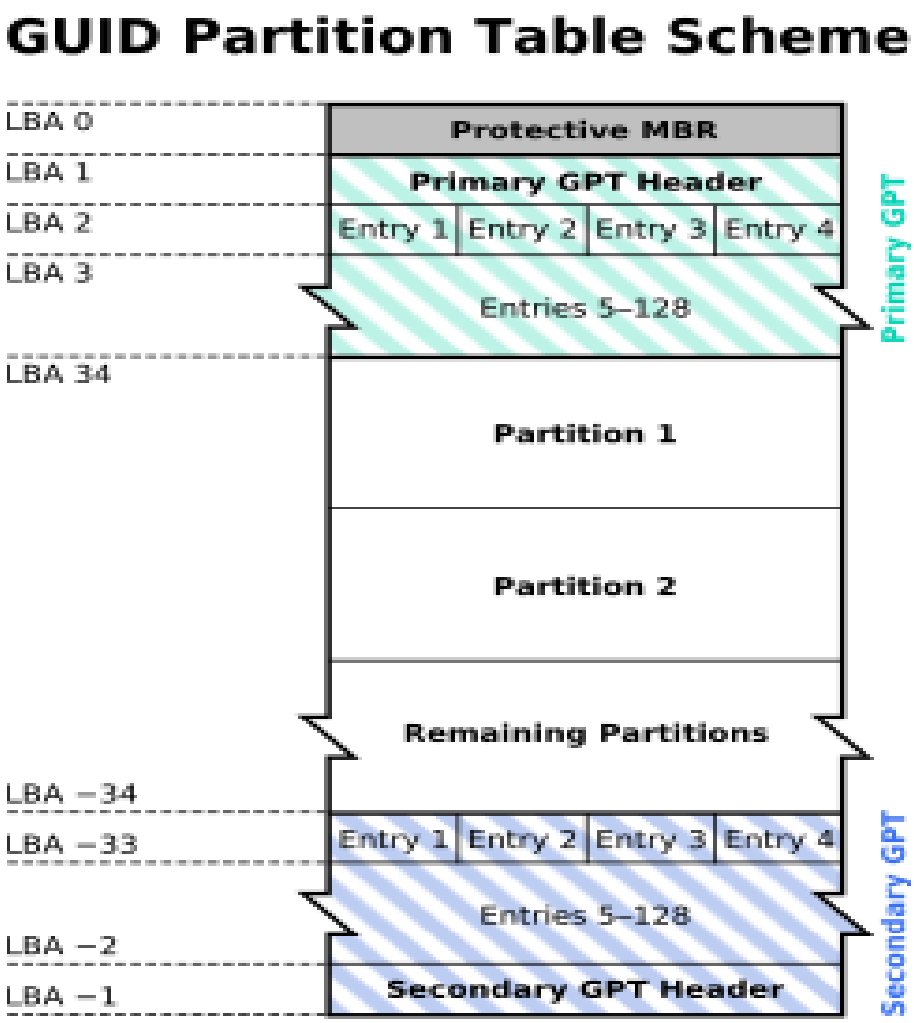

Рисунок 2.6 - Структура таблицы разделов GPT

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.

2.8 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 справочным пособием, является пособие [3, раздел 2].

2.9 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.

2.10 Лабораторная работа по теме №2

Учебная цель данной лабораторной работы - закрепление теоретических знаний и получение практических навыков по теме 2 «*BIOS, UEFI и загрузка ОС*».
Основным учебным пособием, необходимым для работы, является [3, раздел 2].
Данный раздел учебного пособия содержит материал, касающийся основных организационных мероприятий, обеспечивающих успешное выполнение работы.
Ограничения, применительно к которым изложен данный методический материал:

- студент выполняет работу в учебном классе кафедны АСУ на компьютерах с установленным ПО ОС УПК АСУ, имеющих базовое ПО BIOS и структуру загрузочных устройств MBR;
- студент имеет flashUSB с установленным ПО GRUB2 и архивом рабочей области пользователя ***upk*** по данной дисциплине;
- студент успешно выполнил лабораторную работу №1 и способен звпустить ОС УПК АСУ, подключить к ней архив, войти в сеанс пользователя ***upk***, найти местоположение учебного материала и запустить на чтение данное пособие.

Исходные организационные мероприятия:

- *студенты разбиваются парами* (можно больше), каждая из которых имеет устройство flashUSB с минимальным объемом ПО, подготовленным преподавателем на лабораторной работе №1;
- *каждой паре доступна* ЭВМ с запущенным ПО ОС УПК АСУ и учебный материал данного руководства;
- *вопросы вариантов* выполнения данной работы разрешаются преподавателем индивидуально.

2.10.1 Установка ПО GRUB на устройство flashUSB

Изучается три подраздела методического пособия [3, подразделы 2.1-2.3].
Изучается подраздел 2.4 методического пособия [3], учитывая возможные особенности исполнения работ в среде пользователя ***upk***.
Выполняется установка ПО GRUB2 на flashUSB пары, на которых оно отсутствует.
Пары разделяются по отдельным компьютерам, запускают ОС УПК АСУ и заполняют личные отчеты по выполненной части работ. Отчет заполняется в плане всех вопросов, изложенных в пособии [3, подразделы 2.1-2.4].
Проводится индивидуальный перезапуск ОС УПК АСУ с целью проверки качества исполнения проведенной части работы.

Замечание

По результатам выполнения этой части работы, все студенты должны иметь личные flashUSB с установленным на них рабочим ПО GRUB2.

2.10.2 Создание аварийного варианта ОС УПК АСУ

Изучается подраздел методического пособия [3, подраздел 2.5].
Выполняется установка ПО аварийного варианта ОС УПК АСУ на flashUSB студента.
Студенты, имеющие персональные компьютеры и желающие проводить на них учебные работы, дополнительно устанавливают аварийный вариант ПО ОС УПК АСУ на диск С: своих ЭВМ.
Заполняется личный отчет по выполненной части работы, в плане всех вопросов, изложенных в пособии [3, подраздел 2.5].

Замечание
Аварийный вариант дистрибутива ОС УПК АСУ не содержит графической оболочки и находится в директории: */usr/lib/upkasu/flashUSB*
Необходимо полностью скопировать содержимое этой директории *в корень личного устройства flashUSB студента*.

2.10.3 Практика настройки файла конфигурации grub.cfg

Изучается подраздел методического пособия [3, подраздел 2.6].
Выполняются эксперименты с настройками файла grub.cfg.
Студенты, установившие аварийный вариант ПО ОС УПК АСУ на свою ЭВМ, выполняют модификацию первого пункта меню в файле grub.cfg, применительно к запуску ОС со своего комьютера, и проводится экспериментальная проверка выполненой работы.
Заполняется личный отчет по выполненной части работы, в плане всех вопросов, изложенных в пособии [3, подраздел 2.6].
Выполняется создание архива и его перенос на личный flashUSB, а также все мероприятия по завершению лабораторной работы №2.

Замечание
Студенты, выполняющие работу на личных ЭВМ, имеющих UEFI, выполняют эту часть работы с преподавателем.

3 Тема 3. Языки управления ОС

Многие теоретические знания, представленные в данном разделе уже использовались при изучении материала предыдущих двух тем или известны студенам из учебного материала других дисциплин. Тем не менее, имеется ряд моментов, которые необходимо хорошо усвоить, прежде чем изучать вопросы, изложенные в последующих темах. Кроме того, для правильного понимания сути многих вопросов, необходим адекватный язык, на котором эту суть можно объяснить. Для этой цели и предназначен учебный материал данного раздела.

Чтобы учебный материал получил должный уровень конкретизации, он опирается на практическую часть задач загрузки ОС УПК АСУ, изложенных в [3] и доступных в виде файла *upk_asu.pdf* на рабочем столе пользователя ***upk***.

3.1 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 ***C++, 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 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.

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. Их современное совместное существование вызвано:

- силой привычки, авторскими правами и рядом корпоративных интересов;
- наличием достаточно большого количеста ПО, написанного ранее на этих языках.

Общая проблематика интерпретаторов - увеличение функциональных возможностей shell влечет:

- увеличение его размера и уменьшение скорости его загрузки;
- повышенный расход оперативной памяти компьютера.

3.2 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 ***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 системного вызова ***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. На рисунке 3.1 показаны ресурсы типового прикладного процесса.

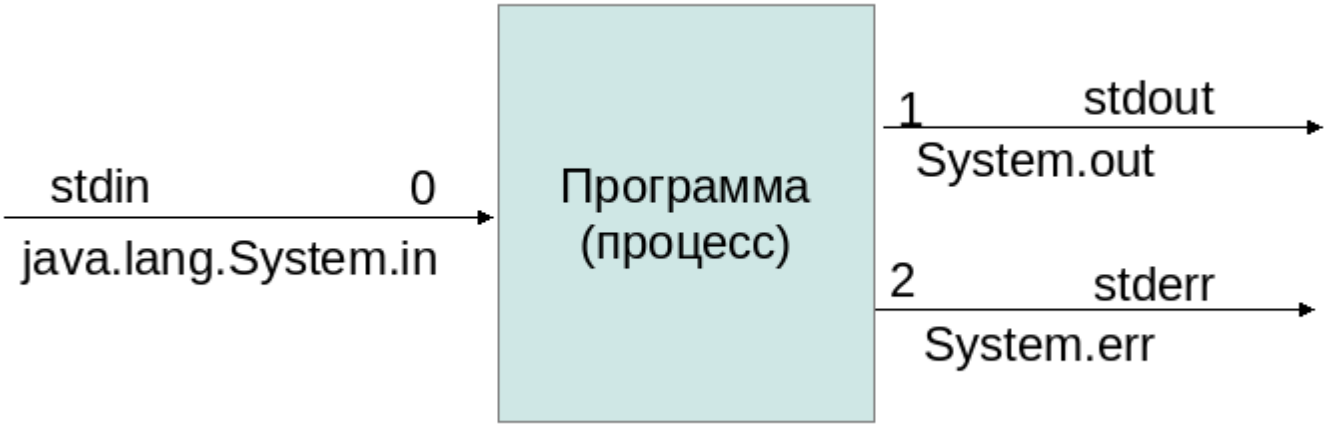

Рисунок 3.1 - Потоки ввода/вывода программы

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

- **один** системный ввод;
- **два** системных вывода.

На уровне файловых дескрипторов, мы имеем:

устройство 0 — устройство ввода;
устройство 1 — устройство нормального вывода программы;
устройство 2 — устройство вывода ошибок.

На уровне языка C, 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(...)**.

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.

Кроме того, sh использует *все 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*.

Интерпретатор sh может использовать текстовые файлы как программы: *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.

3.3 Среда исполнения 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, определяется условиями видимости *той части файловой системы*, которая соответствует *пользователю*, запустившему shell. В течении работы и запуска, sh использует следующие конфигурационные файлы:

```
/etc/profile
$HOME/.profile
/tmp/sh*
/dev/null
```

Shell использует следующие *переменные среды*:

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 каж-дым 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(5)] - это *набор пар (имя, значение)*, который передается выполняемой программе так же, как и обычный список аргументов.
Shell взаимодействует с окружением несколькими способами:

- при запуске, shell просматривает окружение и создает переменную (*ключе-вой параметр*) для каждого указанного имени, придавая ей соответствующее значение;
- если *изменить значение* какой-либо переменной или создать новую, это не окажет никакого влияния на окружение, если не будет использована команда **export** для увязывания переменной shell с окружением (см. также **set -a**);
- *переменную 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, указанные в команде **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*.

Если установлен флаг **-k**, то все переменные, получившие значение в командой строке, помещаются в окружение команды, даже если они записаны после команды. В следующем примере, показанном на рисунке 3.2, приведена работа команды ***echo***, предназначенной для вывода строк на консоль:

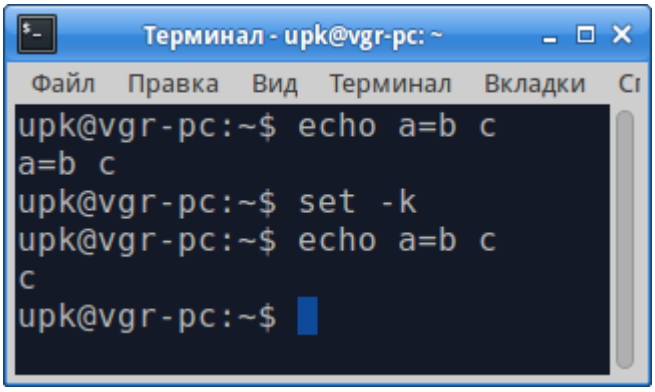

Рисунок 3.2 — Использование ключа ***-k*** в языке shell

3.4 Командная строка: опции и аргументы

Как было показано ранее, **первое слово** в строке shell воспринимает *как команду*, а остальные слова — *как аргументы команды*.

Пример, приведенный на рисунке 3.2, показывает, что это не всегда так:

- 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 выполняю-

тся 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, символов подчеркивания, начинающаяся с буквы или подчеркивания;
- *параметр* - это имя, цифра или любой из символов **, @, #, ?, -, $, !*.

Перейдем к рассмотрению *опций* и *аргументов*. В общем случае, 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:

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*.

-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

интерактив-ного shell'a).
Сигнал прерывания (2) перехватывается и игнорируется, поэ-тому выполнение системной функции wait (2) может быть прервано.
В любом случае, сигнал выхода (3) игнорируется.

-r

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

Замечание

Описание остальных флагов и аргументов приведено в описании команды **set**.

3.5 Переменные shell

Все переменные shell, включая рассмотренные выше *переменные среды*, называются *параметрами*:

- 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.

Пример *вывода позиционных параметров*, часть из которых установлено автоматически, показан сценарием рисунка 3.3, а результат вывода - рисунком 3.4.

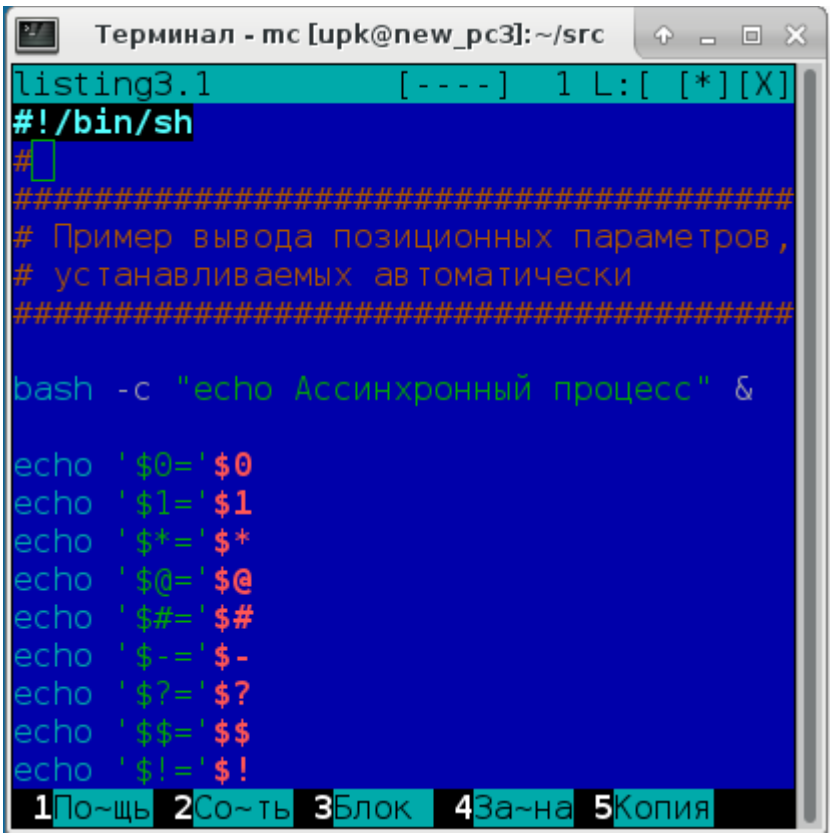

Рисунок 3.3 — Сценарий вывода позиционных параметров

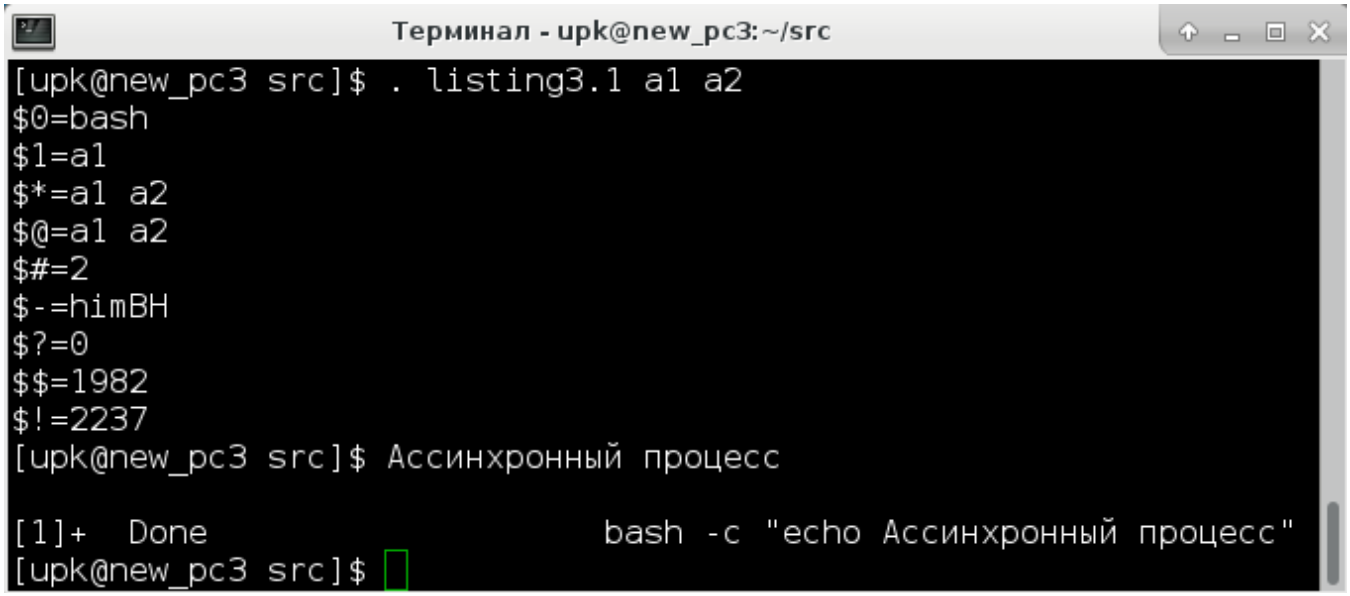

Рисунок 3.4 — Результат вывода сценария

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'a завершается. Если слово опущено, то выдается сообщение "параметр:parameter null or not set".

${параметр:+слово}

Если параметр ***не пуст***, 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***.

3.6 Специальные символы и имена файлов

Shell интерпретирует команды и аргументы команд как слова.
Следующие символы, если они не экранированы, завершают предыдущее слово:

```
;  &  (  )  |  ^  <  >  пробел  табуляция  перевод_строки
```

Эти символы могут экранироваться *одинарными* или *двойными кавычками*.

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

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

В командах, работающих с именами файлов, возможно использование *шаблонов*:

- ***** 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 **[** 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.

Замечание

Не следует надеяться на интуицию!
Обязательно следует проверить результаты вывода шаблонов в командной строке терминала.

3.7 Стандартный ввод/вывод и переадресация

Типичный процесс, показанный ранее на рисунке 1.1, читает данные с клавиатуры (дескриптор файла 0) и выводит данные на экран терминала (дескрипторы файлов 1 и 2).

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, затем для каждой команды конвейера, затем 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.

Замечание

Следует очень внимательно работать с перенаправлениями, поскольку они являются разделителями прав доступа. Например, две команды, подключающие архив студента к рабочей области ОС УПК АСУ, являются разными по правам доступа и не обеспечивают нужный результат:

```
gzip -cdvk /run/media/FC99-4744/asu64upk/themes/os-home.ext4fs.gz > \
     /run/basefs/asu64upk/themes/os-home.ext4fs

sudo gzip -cdvk /run/media/FC99-4744/asu64upk/themes/os-home.ext4fs.gz > \
     /run/basefs/asu64upk/themes/os-home.ext4fs
```

Хотя вторая команда подвергается воздействию команды *sudo*, ее действие распространяется только до оператора перенаправления, что не позволяет записывать результат на защищенные устройства.

Чтобы устранить указанный недостаток, используются отдельные сценарии *from-gzip* и *to-gzip*, содержимое которых показано на рисунках 3.5 и 3.6. Применение команды *sudo* к сценарию обеспечивает командам нужный уровень привилегий.

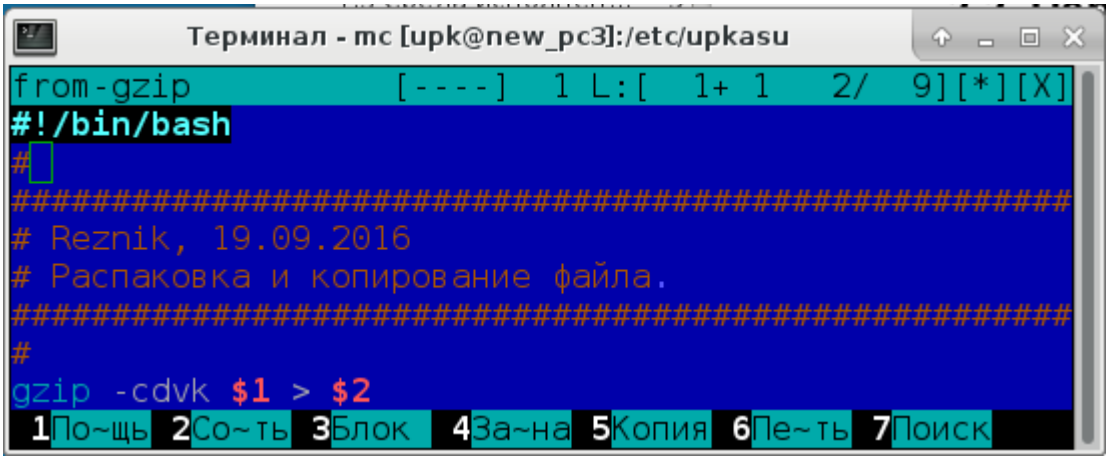

Рисунок 3.5 — Сценарий распаковки архива темы

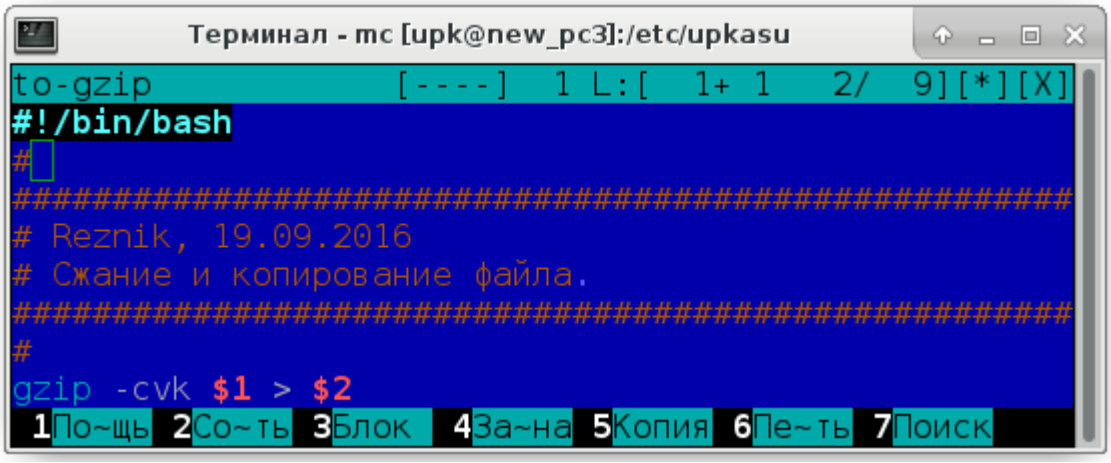

Рисунок 3.6 — Сценарий сжатия рабочей области и записи его в нужное место

3.8 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*.

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

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

Конвейер - это последовательность команд, разделенных знаком |.

При этом:

Стандартный вывод всех команд, кроме последней, направляется посред-ством системного вызова *pipe(2)* на стандартный ввод следующей команды конвейера.

Каждая команда выполняется как ***самостоятельный процесс***.

shell ожидает завершения последней команды. Ее код завершения станови-тся ***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***.

Замечание

Ранее отмечено, что двойные и одинарные кавычки используются shell для *экранирования последовательности слов*, с целью рассмотрения этой последовательности как *отдельный аргумент*.

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. Результат работы данного примера приведен на рисунке 3.7.

Замечание

Такого же результата можно добиться командой:

```
abc=$(ls -l);
```

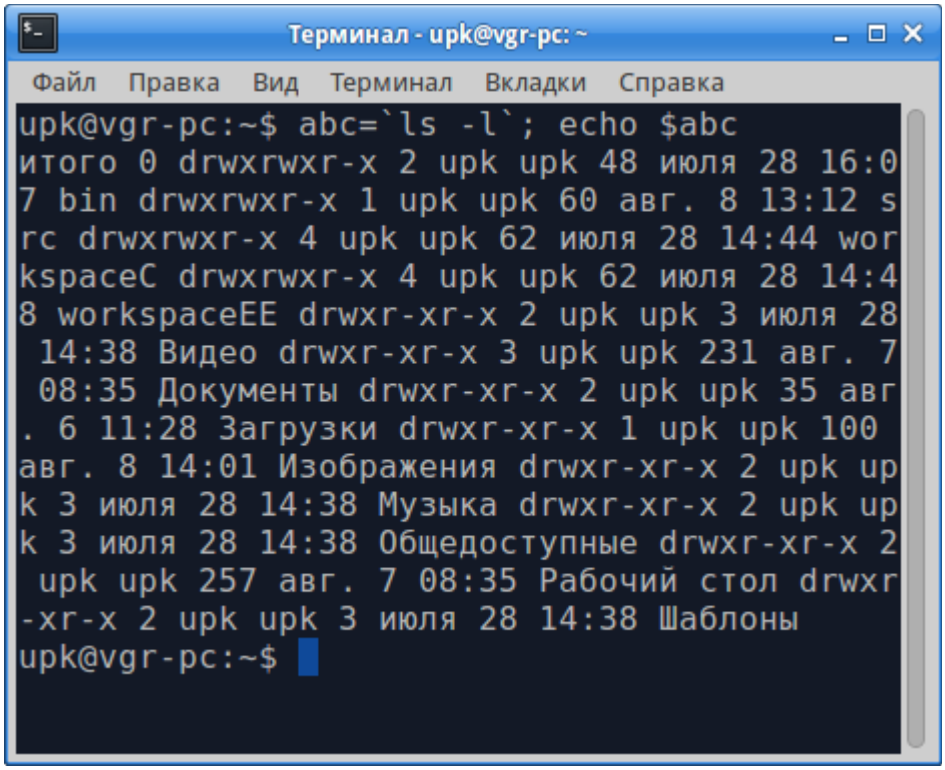

Рисунок 3.7 — Пример использования обратных кавычек

3.9 Сценарии

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

Язык shell содержит следующие *управляющие конструкции*.

```
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 устанавливается 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 shell как ***ключевые***, если они являются первым словом команды и не экранированы:

```
if  then  elif  else  fi
case  in  esac
for  while  until  do  done
{  }
```

Изучив понятие сценария, закрепим предыдущий учебный материал двумя примерами, решающими задачи: чтение и подключение архива студента, а также запись архива на личный flashUSB.
Текст сценариев показан на рисунке 3.8 и листингах 3.1-3.2.
Рассмотрение примеров начнем со *сценария экспорта параментров* ПО УПК АСУ.

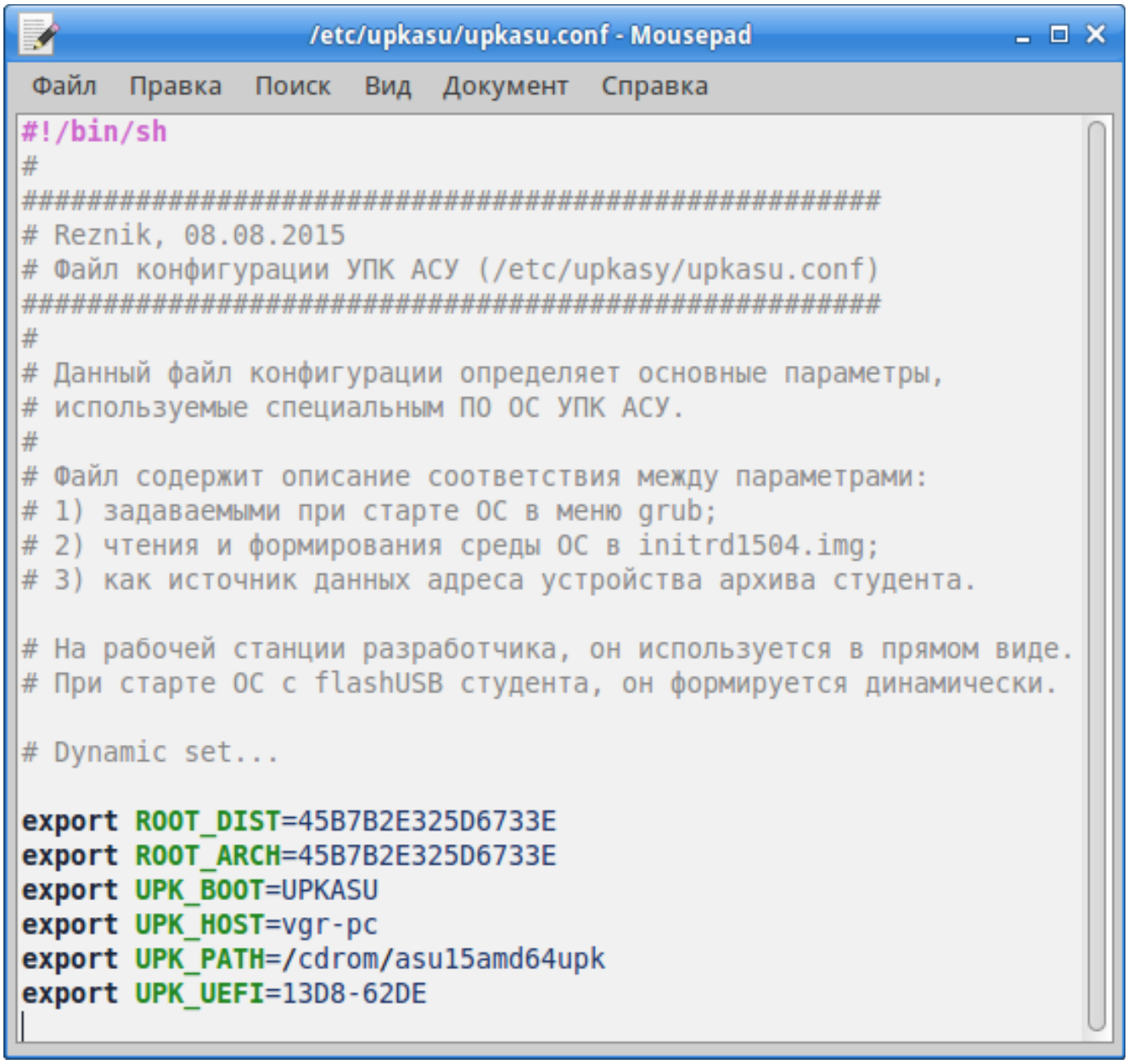

Рисунок 3.8 — Сценарий экспорта параметров ПО УПК АСУ

Листинг 3.1 — Сценарий mount-upk.sh — подключение архива студента

```
#!/bin/bash

############################
# Специально для ОС УПК АСУ
# Reznik, 02.09.2016
# Подключение темы обучения
############################

# Используем функцию для вывода сообщения и продолжения работы программы:
if_msg()
{
      echo -e  "$@"
      echo -en "Нажми клавишу$green Enter$gray для продолжения ... "
      read aa
}
# Используем функцию для вывода сообщения и выхода из программы:
if_exit()
```

```
{
      echo -e  "$@"
      echo -e  "------------------------------------------------------------"
      echo -en "Нажми клавишу$green Enter$gray для завершения работы программы ...
"
      read aa
      exit 0
}

#-----------------------------------
# Подготовительная часть
#-----------------------------------
. /etc/upkasu/upk.colors
echo -e "${green}Проверяем пользователя!$gray"
      [ "$USER" != "upk" ] || \
      if_exit "Ползователь upk не может устанавливать тему обучения!!!"

echo "Читаем парметры окружения..."
. /etc/upkasu/upkasu.conf

echo  "Останавливаем  все  процессы  пользователя  upk,  подключенные  к  директории
/home/upk..."
for xx in $(sudo lsof -t /home/upk/); do
      [ -d "/proc/$xx" ] || continue
      echo -e "Удаляю процесс$red $xx $gray"
      sudo kill -9 "$xx"
done

echo "Проверяем наличие монтирования к директории /home/upk ..."
      x1=$(mount | grep "/home/upk")
      [ "x$x1" == "x" ] || sudo umount -f /home/upk

      x1=$(mount | grep /home/upk | cut -d ' ' -f 1 )
      [ "x$x1" == "x" ] || if_exit "${red}Не могу отмонтировать файл${gray}: $x1"
#-----------------------------------
# Определяем устройства архива:
xdev=$(blkid -U "$ROOT_ARCH")
echo "Проверяем монтирование устройства архива..."
[ "x$xdev" != "x" ] || \
      if_exit "${red}Подмонтируйте$gray устройство архива... и попробуйте снова!.."

echo "Находим директорию архива..."
xdir=$(mount | grep "$xdev" | cut -d ' ' -f 3 )
[ "x$xdir" != "x" ] || if_exit "${red}Не могу найти$gray директорию архива... "
if [ "x$xdir" == "x/" ]; then
      xdir=""
fi

echo "Находим директорию дистрибутива..."
wdir="${UPK_MOUNT}"
[ "x$wdir" != "x" ] || if_exit "${red}Не могу найти$gray директорию дистрибутива...
"
if [ "x$wdir" == "x/" ]; then
      wdir=""
fi

xdir="${xdir}/${UPK_PATH}/themes"
[ -d "$xdir" ] || if_exit "${red}Отсутствует$gray директория архива... "
wdir="${wdir}/${UPK_PATH}/themes"
[ -d "$wdir" ] || if_exit "${red}Отсутствует$gray директория дистрибутива... "

#-----------------------------------
# Выбираем тему обучения
echo "Читаем список доступных тем обучения..."
[ "$xdir" == "$wdir" ] && fl=home.ext4fs || fl=home.ext4fs.gz
```

```
xlist=$(ls "$xdir" | grep $fl )
[ "x$xlist" != "x" ] || if_exit "${red}Нет тем${gray}, доступных для подключения...
"

echo -e "${green}Список доступных тем$gray обучения..."
echo "--------------------------------"
for xx in ${xlist}; do
      x1=$(echo $xx | cut -d '-' -f 1 )
      echo -e "${yellow}${x1}$gray"
done
echo "--------------------------------"

echo -en "${green}Введи имя учебной темы$gray для подключения: "
read aa
export THEME="$aa"

echo "Проверяем наличие архива..."
xarch="${xdir}/${THEME}-$fl"
[ -f "$xarch" ] || if_exit "${red}Отсутствует файл архива${gray}: ${xarch}"
warch="${wdir}/${THEME}-home.ext4fs"

#-----------------------------------
# Копируем тему обучения

echo "Запоминаю тему обучения..."
echo "export UPK_THEME=$THEME" > ~/.upk_theme
if [ "$xarch" != "$warch" ]; then
      echo "Копирую файл архива в рабочую область..."
###   sudo cp -v "$xarch" "$warch"
      sudo /etc/upkasu/from-gzip "$xarch" "$warch"
fi

#-----------------------------------
# Монтирование рабочей области:

echo -e "${green}Монтируем$gray файловую систему пользователя$yellow upk$gray \
в директорию$blue /home/upk$gray ..."

sudo mount -t ext4 "$warch" /home/upk -o loop
[ "$?" == "0" ] || if_exit "${red}Не могу смонтировать$gray $warch на /home/upk"

if [ -f "/home/upk/.upk_theme" ]; then

echo -e "\n"\
"Тема$blue $THEME -$green успешно подлючена...$gray \n"\
"Вам следует выйти из сессии пользователя$yellow asu$gray и подключитья \n"\
"как пользователь$yellow upk ${gray}!"

      if_exit " "
fi

if_exit "Тема $THEME - подлючена (Необходима дополнительная проверка!!!)"
```

Листинг 3.2 — Сценарий mount-upk.sh — подключение архива студента

```
#!/bin/sh
#
################################################################
# Reznik, 02.09.2016
# Копирование рабочей области пользователя upk на flashUSB
#
# Файл сценария: /etc/upkasu/copy-to-flash.sh
################################################################
```

```
#
# Данный сценарий обеспечивает копирование рабочей области
# темы обучения на личный flashUSB студента.

# Используем функцию для вывода сообщения и выхода из программы:
if_exit()
{
      echo -e "$@"
      read -p "Нажми клавишу Enter..." aa
      exit 0
}

# Читаем парметры окружения:
. /etc/upkasu/upkasu.conf
. /etc/upkasu/upk.colors
. ~/.upk_theme

echo "##########################################"
echo -e "${red}Копирование архива на flashUSB студента...${gray}\n"
[ "$ROOT_ARCH" != "$ROOT_DIST" ] || \
      if_exit "${green}Рабочая область и архив - одно и тоже устройство!!!${gray}"

# Находим устройства архива и дистрибутива ОС:
xdev=$(blkid -U "$ROOT_ARCH")
[ "x$xdev" != "x" ] || \
      if_exit  "${yellow}Необходимо  вставить  устройство  архива  и  попробовать
снова!..${gray}"
xdir=$(mount | grep "$xdev" | cut -d ' ' -f 3 )
[ "x$xdir" != "x" ] || \
      if_exit  "${yellow}Подмонтируйте устройство архива... и попробуйте снова!..$
{gray}"
[ "x$xdir" != "x/" ] || xdir=""
dira="${xdir}/${UPK_PATH}/themes"
[ -d "$dira" ] || \
      if_exit "Не могу найти директорию архива... Обратитесь к преподавателю!.."

# Проверяем монтирование рабочей области:
wdir="$UPK_MOUNT"
[ "x$wdir" != "x/" ] || wdir=""
fa="${wdir}/${UPK_PATH}/themes/${UPK_THEME}-home.ext4fs"
xx=$(mount | grep "$fa" )
[ "x$xx" == "x" ] || \
      if_exit "${yellow}Сначала отключите тему, а потом - архивируйте!${gray}"

sudo /etc/upkasu/to-gzip "$fa" "${dira}/${UPK_THEME}-home.ext4fs.gz" || \
      if_exit  "${yellow}Не  могу  перенести  архив...  Обратитесь  к  преподавателю!..$
{gray}"

# Закончили копирование архива:
      if_exit   "${green}Перенесено:$blue   из   ${fa}$red   в   ${dira}/${UPK_THEME}-
home.ext4fs.gz${gray}"
```

Кроме управляющих конструкций, shell содержит ряд встроенных (*специальных*) команд.

Замечание

Если не оговорено иное, *команды* выводят результаты в файл с *дескриптором 1*.

: Пустая команда. Возвращает нулевой код завершения.

. файл
Shell читает и выполняет команды из файла, затем возобновляется чтение со стандартного ввода; при поиске файла используется значение переменной **PATH**.

break [n]
Выйти из внутреннего ***for*** или ***while*** цикла; если указано **n**, то выйти из **n** внутренних циклов.

continue [n]
Перейти к следующей итерации внутреннего ***for*** или ***while*** цикла; если указано **n**, то перейти к следующей итерации **n**-ого цикла.

cd [каталог]
Сделать текущим заданный каталог. Если каталог не указан, используется значение переменной **HOME**. Переменная **CDPATH** определяет список поиска каталога. По умолчанию этот список пуст (то есть поиск производится только в текущем каталоге). Если каталог начинается с символа **/**, список поиска не используется.

echo [аргумент ...]
Выдать аргументы на стандартный вывод, разделяя их пробелами [см. также echo(1)].

eval [аргумент ...]
Выполнить команду, заданную аргументами ***eval***.

exec [аргумент ...]
Сменить программу процесса: в рамках текущего процесса команда, заданная аргументами exec, заменяет shell. В качестве аргументов могут быть указаны спецификации ввода/вывода и, если нет никаких других аргументов, будет лишь переназначен ввод/вывод текущего shell'a.

exit [код_завершения]
Завершить выполнение shell'a с указанным кодом. При отсутствии аргумента код завершения определяется последней выполненной командой. Чтение символа конца файла также приводит к завершению shell'a.

export [переменная ...]
Заданные переменные отмечаются для автоматического экспорта в окружение выполняемых команд. Если аргументы не указаны, выводится список всех экспортируемых переменных. Имена функций не могут экспортироваться. Имена переменных, экспортированных из родительского shell'a, выдаются только в том случае, если они были экспортированы и из текущего shell'a.

getopts
Используется в shell-процедурах для поддержания синтаксических стандартов [см. intro(1)]; разбирает позиционные параметры и проверяет допустимость задания опций [см. getopts(1)].

hash [-r] [имя_команды ...]

Для каждого из указанных имен_команд определяется и запоминается маршрут поиска - место в списке поиска, где удалось найти команду. Опция -r удаляет все запомненные данные. Если не указан ни один аргумент, то выводится информация о запомненных командах: **hits** - количество обращений shell'а к данной команде; **cost** - объем работы для обнаружения команды в соответствии со списком поиска; command - полное имя команды. Звездочкой в колонке hits помечаются команды, маршрут поиска которых будет перевычисляться после смены текущего каталога [см. cd(1)]; расходы на перевычисление учитываются в колонке cost.

newgrp [аргумент ...]
Выполяет регистрацию пользователя в новой группе [man 1 newgrp].

pwd Выводит имя текущего каталога. [см. pwd(1)].

read [переменная ...]
Со стандартного ввода читается одна строка и делится на слова с учетом разделителей, перечисленных в значении переменной IFS (обычно это пробел или табуляция); первое слово присваивается первой переменной, второе - второй и т.д., причем все оставшиеся слова присваиваются последней переменной. Исходная строка имеет продолжение, если в конце ее стоит последовательность \перевод_строки. Символы, отличные от перевода строки, также могут быть экранированы с помощью \, который удаляется перед присваиванием слов. Возвращается нулевой код завершения, если только не встретился конец файла.

readonly [переменная ...]
Указанные переменные отмечаются как доступные только на чтение. Присваивания таким переменным трактуются как ошибки. Если аргументы не указаны, то выводится информация обо всех переменных, доступных только на чтение.

return [код_завершения]
Выйти из функции с указанным кодом_завершения. Если аргумент опущен, то код завершения наследуется от последней выполненной команды.

set [-a] [-e] [-f] [-h] [-k] [-n] [-t] [-u] [-v] [-x] [--] [аргумент …]]

-a Экспортировать переменные, которые изменяются или создаются, в окружение.

-e Выйти из shell'а, если какая-либо команда возвращает ненулевой код завершения.

-f Запретить генерацию имен файлов.

-h Определить и запомнить местоположение всех команд, входящих в тело функции, во время ее определения, а не во время выполнения.

-k Поместить в окружение команды все переменные, получившие значение в командной строке, а не только те, что предшествуют имени команды.

-n Читать команды, но не выполнять их.

-t Выйти из shell'а после ввода и выполнения одной команды.

-u Рассматривать подстановку параметров, не получивших значений, как ошибку.

-v Выводить исходные для shell'а строки сразу после их ввода.

-x Выводить команды и их аргументы непосредственно перед выполнением.

-- Не изменяет флаги. Полезно использовать для присваивания позиционному параметру $1 значения -:

```
set -- -
```

При указании + вместо - перечисленные выше режимы выключаются. Описанные флаги могут также использоваться при запуске shell'а. Набор текущих флагов есть значение переменной $-. Следующие за флагами аргументы будут присвоены позиционным параметрам $1, $2 и т.д. Если не заданы ни флаги, ни аргументы, выводятся значения всех переменных.

shift [n]
Позиционные параметры, начиная с (n+1)-го, переименовываются в $1 и т.д. По умолчанию n=1.

test Вычислить условное выражение [см. test(1)].

times Вывести суммарные времена пользователя и системы, затраченные на выполнение процессов, запущенных данным shell'ом.

trap [имя_команды] [n] ...
Команда с указанным именем будет прочитана и выполнена, когда shell получит сигнал(ы) n. Заметим, что имя_команды обрабатывается при установке прерывания и при получении сигнала. Команды выполняются в порядке номеров сигналов. Нельзя установить обработку прерывания по сигналу, игнорируемому данным shell'ом. Попытка установить обработку прерывания по сигналу 11 (выход за допустимые границы памяти) приводит к ошибке. Если имя_команды опущено, то для прерываний с указанными номерами n восстанавливается первоначальная реакция. Если имя_команды есть пустая строка, то этот сигнал будет игнорироваться shell'ом и вызываемыми им программами. Если n равно 0, то указанная команда выполняется при выходе из shell'а. Trap без аргументов выводит список команд, связанных с каждым сигналом.

type [имя ...]
Для каждого имени указывается, как оно будет интерпретироваться при использовании в качестве имени команды.

ulimit [размер_в_блоках]
Установить максимальный размер_в_блоках (по 1 Кб) тех файлов, в которые пишут данный shell и его потомки (читать можно файлы любого размера) [см. ulimit(1)]. Если размер не указан, выдается текущий лимит. Каждый пользователь может уменьшить

```
собственный лимит, но только суперпользователь мо-
жет его увеличить.

umask  [nnn]
     Пользовательская маска создания файлов  становится
     равной nnn (восьмеричное) [см. umask(1)]. Если nnn
     опущено, выдается текущее значение маски.

unset  [имя ...]
     Для каждого указанного имени удалить соответствую-
     щую переменную или функцию. Переменные PATH,  PS1,
     PS2, MAILCHECK и IFS не могут быть удалены.

wait  [идентификатор_процесса]
     Ждать завершения указанного  фонового  процесса  и
     вывести код его завершения. При  отсутствии  аргу-
     мента ждать завершения всех активных фоновых  про-
     цессов. В этом случае код завершения  будет  нуле-
     вым [см. wait(1)].
```

3.10 Фоновый и приоритетный режимы

В интерактивном *режиме*, ***shell*** взаимодействует с конкретным *пользователем* посредством *консоли* (терминала):

- пользователь в консоли набирает (редактирует) 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.

Например,

```
$   ls -l & «Ввод»
[1] 534
… - список файлов текущей директории
$
```

Для просмотра списка запущенных заданий используется команда **jobs**. Например,

```
$  mousepad .upk_theme &
```

```
[1]  547
$  cat *.c > myprogs &
[2]  548
$  jobs
[1]  + Running lpr intro
[2]  -  Running cat *.c > myprogs
$
```

здесь знак ***плюс*** означает выполняемое в данный момент задание, а знак ***минус*** — задание, ожидающее выполнения.

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

```
$  fg %2
cat *.c > myprogs
$
```

3.11 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***.
Например,

```
$  jobs
[1]  + Running lpr intro
[2]  -  Running cat *.c > myprogs
$  kill %2
$
```

или, тоже самое:

```
$  kill 548
$
```

3.12 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*.

3.13 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]
```

Замечание

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

3.14 Лабораторная работа по теме №3

Цель лабораторной работы №3 — практическое закрепление учебного материала по теме «Языки управления ОС».
Метод достижения указанной цели — чтение учебного материала, изложенного в первом разделе данного пособия и выполнение указанных в тексте команд в окне терминала.

Чтобы успешно выполнить данную работу, студенту следует:

- *запустить ОС УПК АСУ*, подключить личный архив и переключиться в сеанс пользователя ***upk***;
- *запустить на чтение* данное пособие и на редактирование личный отчет;
- *открыть одно или несколько окон терминалов*, причем хотя бы в одном окне терминала открыть Midnight Commander, для удобства работы с файловой системой ОС;
- *приступить к выполнению работы*, последовательно пользуясь рекомендациями представленных ниже подразделов.

Замечание

Многие команды ОС студенту еще не известны, поэтому следует:

- для вывода на консоль руководства по интересуемой команде, использовать: ***man имя_команды***;
- для выяснения сущесвования команды, ее доступности и местоположения, использовать: ***command -v имя_команды***;
- для уточнения правил запуска конкретной команды, можно попробовать один из вариантов: ***команда --help*** или ***команда -h*** или ***команда -?***.

3.14.1 Среда исполнения программ

Прочитайте и усвойте материал подразделов 3.1-3.3 данного пособия.
Повторно, в подразделе 532, разберитесь с потоками ввода/вывода программы.
В подразделе 3.3:

- выполните в терминале команду ***env***; изучите содержимое вывода и отразите в отчете;
- выведите содержимое основных переменных среды командой: ***echo $имя***;
- выполните команду: ***echo $UPK_THEME***;
- просмотрите содержимое файла командой: ***cat ./.upk_theme***;
- выполните команду: ***. ./.upk_theme***;
- выполните команду: ***echo $UPK_THEME***;
- выполните команду: ***unset UPK_THEME***;
- выполните команду: ***echo $UPK_THEME***;
- результаты исследования отразите в отчете.

3.14.2 Переменные, опции и аргументы командной строки

Прочитайте и усвойте материал подразделов 3.4-3.6 данного методического пособия.
Повторно прочитайте подраздел 3.5.
Просмотрите и выполните сценарий *listing3.1*, как показано на рисунках 3.3-3.4.
Разберитесь и самостоятельно проверьте работу подстановок в параметрах и отразите результаты в отчете.
В подразделе 3.6, исследуйте работу шаблонов:

выполняйте команды: ***ls шаблон***;

результаты исследования отражайте в отчете.

3.14.3 Стандартный ввод/вывод и переадресация

Прочитайте и усвойте материал подразделов 3.7 данного методического пособия.
Повторно изучите переадресацию и выполните:

```
echo 'Текст 1' > файл
cat  ./файл
echo 'Текст 2' >> файл
cat  ./файл
echo 'Текст 2' > файл
cat  ./файл
```

результаты исследования отразите в отчете.

3.14.4 Программные каналы и сценарии

Прочитайте и усвойте материал подразделов 3.8-3.9 данного методического пособия.
Перечитайте подраздел 3.8 и выполните демострационные примеры.
Перечитайте часть подраздела 3.9, касающуюся управляющих операторов, и выполните демострационные примеры, показанные на рисунке 3.8 и листингах 3.1, 3.2.
Результаты опишите в отчете.
Перечитайте оставшуюся часть подраздела 3.9 и выполните интересные вам команды.
Результаты опишите в отчете.

3.14.5 Работа с процессами и заданиями среды

Прочитайте и усвойте материал подразделов 3.10-3.13 данного методического пособия.
Повторно перечитайте и выполните исследовательские действия по учебному материалу подразделов 3.10-3.12.
Отразите результаты исследований в отчете.
Проведите создание архива и запись его на личный flashUSB.
Попробуйте командами, описанными в подразделе 3.13, перезапустить и выключить компьютер.
Отразите результаты исследований в отчете.

3.14.6 Сценарии ПО GRUB

Запустите на просмотр файл *upk_asu.pdf*.
Перечитайте его подразделы 2.5 и 2.6.
Опишите в отчете основные различия языка shell и языка GRUB.
Проведите создание архива и запись его на личный flashUSB.
Выключите компьютер и завершите лабораторную работу.

4 Тема 4. Управление файловыми системами ОС

Данный учебный материал предполагает, что предыдущие темы студентом изучены и закреплены на практике тремя лабораторными работами. Имея полученные знания, можно провести нужный уровень детализации в изучении дисциплины. Здесь, такая детализация проводится для темы «Управление файловыми системами ОС».

4.1 Устройства компьютера

Впервую очередь, термин *устройства компьютера* ассоциируются с аппаратурой ЭВМ, что в общем случае так и есть. Но в предмете нашей дисциплины, аппаратура компьютера не является непосредственно доступной для ползователя или программ, работающим в пользовательском режиме. Как ранее было отмечено:

- *доступ к устройствам* ЭВМ возможен только через ядро ОС;
- *все устройства* ЭВМ имеют отображение в виде имен в ФС ОС.

С другой стороны, устройства компьютера являются *ресурсами ОС*, которые управляются ядром ОС.
С третьей стороны, *сами файловые системы ОС* располагаются на устройствах компьютера.

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 соответсвующих файловых систем; эти имена соответствуют вершинам файловых систем (ФС);
- *псевдоустройство с именем rootfs* является корнем *виртуальной файловой системы* (*VFS — Virtual File System*).

Когда GRUB загрузил ядро ОС и передал ему файл с временной файловой системой, то после инициации внутренних параметров ядра проводится *монтирование (подключение) корневой файловой системы*. Это подключение (монтирование) проводится проводится самим ядром ОС относительно *внутренней точки ядра с именем* ***rootfs***.
Далее, ядро ОС создает первый процесс, которым является интерпретатор shell. Он, от имени пользователя ***root*** (UID=0) и группы GID=-1, начинает выполнять сценарий **/init**. Поскольку пользователь ***root*** работает *в режиме пользователя*, то он воспринимает точку ядра ***rootfs*** как /.
В процессе выполнения сценария **/init**, к директориям корневой ФС монтируются *другие файловые системы*.

На рисунке 4.1 выведены типы файловых систем, поддерживаемых текущим сеансом ОС УПК АСУ. Все *псевдоустройства ядра ОС* обозначены как *nodev*.

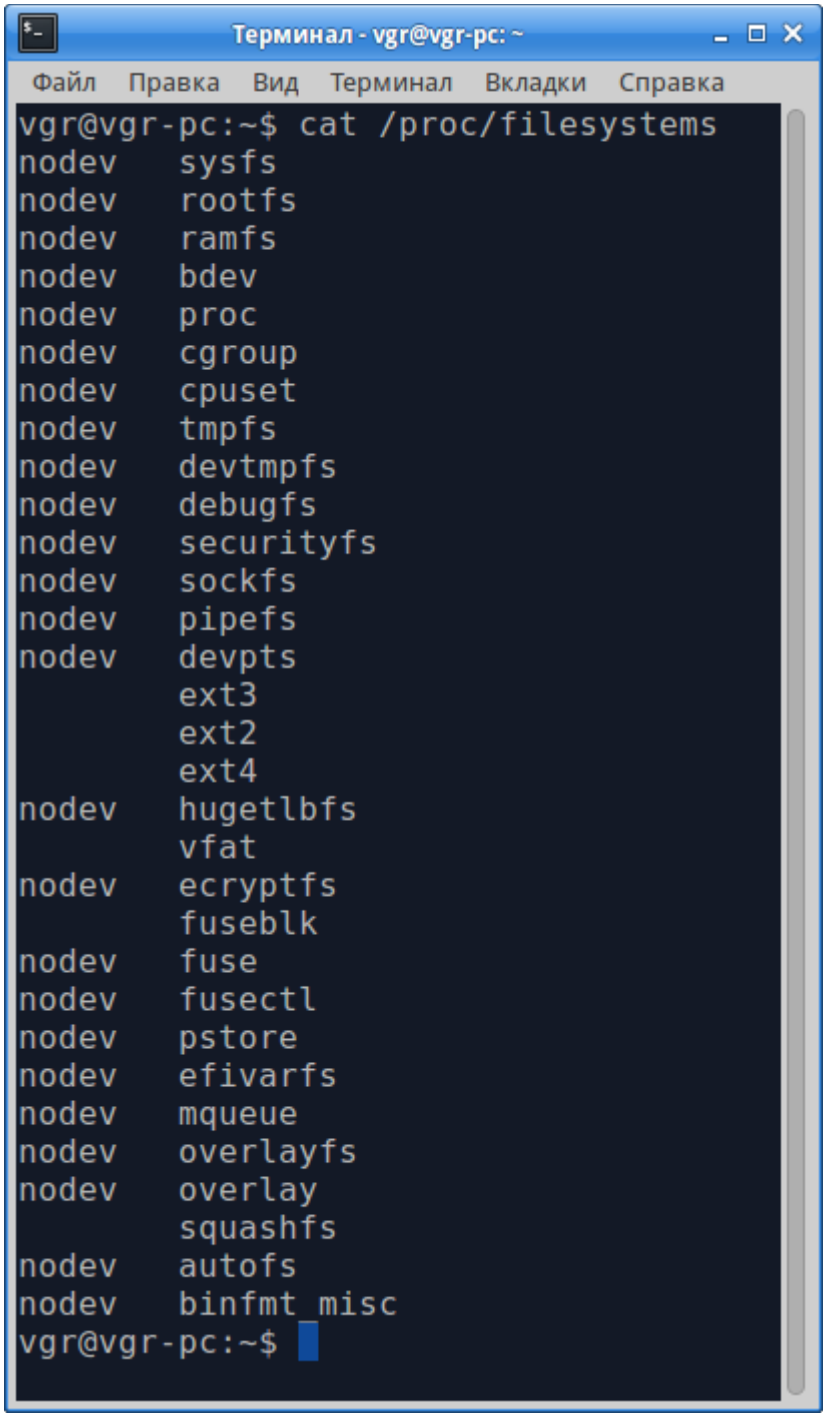

Рисунок 4.1 — Список файловых систем ядра ОС УПК АСУ

На рисунке 4.2 представлен вывод части сообщений команды ***mount***. Хорошо видно, монтирование некоторых псевдоустройств на директории корневой ФС.

По традиции, все ОС UNIX и Linux *отображают имена файлов устройств* в директорию ***/dev*** корневой файловой системы. Для устройств, точкой монтирования ядра ОС является ***udev***, что также хорошо видно из рисунка 4.2.

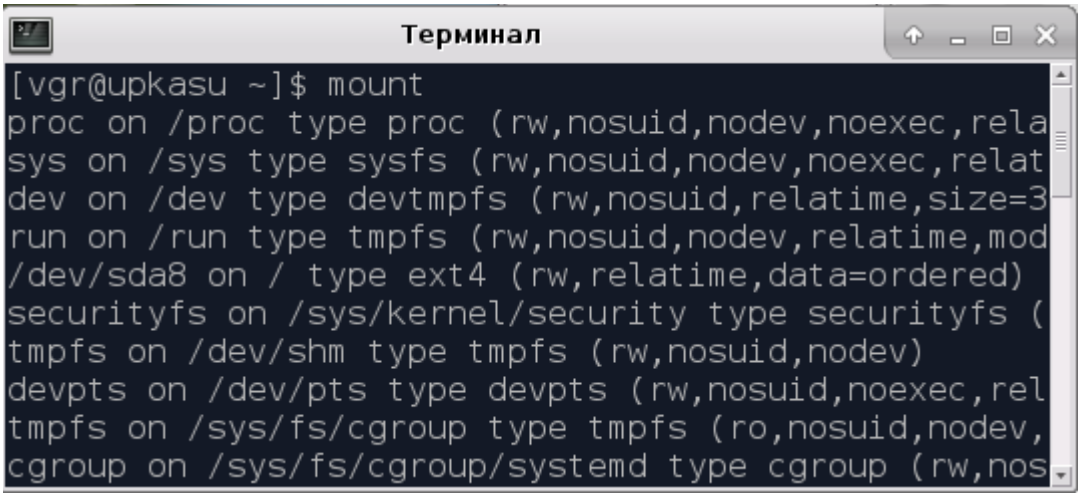

Рисунок 4.2 — Примеры монтирования псевдоустройств ядра

Общая семантика слова *устройство* соответствует семантике английского слова *device*. Реальные файловые системы ОС, на которых долговременно хранится информация, находятся на физических устройствах ЭВМ. Поскольку физические утройства типа винчестера разбиваются на разделы, то *реальные ФС создаются в разделах блочных устройств*. Именно разделы блочных устройств *монтируются к директориям VFS*.

В терминологии ядра, *имя устройства*, которое отображается в директории ***/dev***, называется ***узел*** (***node***).
NODE - *узел* - специальная *именованная структура*, отображаемая в директории ***/dev***, создаваемая для связи ядра ОС с физическим устройством ЭВМ.

Замечание

Чтобы ядро ОС могло работать с физическими устройствами ЭВМ, *узлы (node)* должны быть созданы заранее и иметь отображение в директории ***/dev***.

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). Например, из рисунка 4.2 видно, что директория ***/dev*** монтирована в точку ядра ОС с именем ***udev***, имеющей тип файловой системы ***devtmpfs***.

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.

Например,

```
    [ -e /dev/console ] || mknod -m 0600 /dev/console c 5 1

    [ -e /dev/null ]    || mknod /dev/null c 1 3
```

В этом примере, для устройства ***консоли*** и устройства ***null***, проверяется наличие узла по его имени, и если узлы отсутствуют, то они создаются.
На рисунке 4.3, с помощью команды ***ls -l*** и утилиты фильтра ***grep***, показан вывод информации об узлах *console* и *null*.

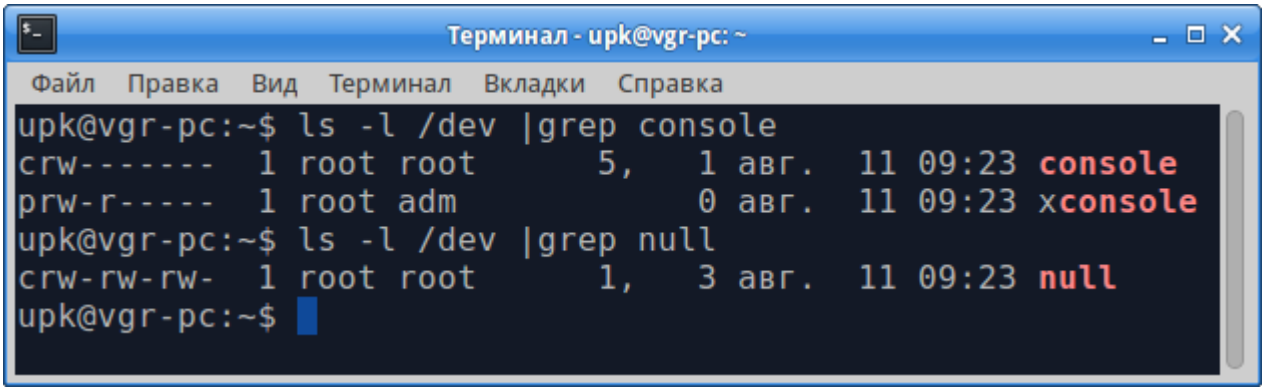

Рисунок 4.3 — Вывод информации об узлах устройств

Замечание

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. Более подробно, эти проблемы обсуждаются в подразделе 1.10 «Три концепции работы с устройствами».
В данном курсе, подробная работа с символьным устройствами не рассматривается, поскольку она всегда имеет свою специфику и детальное знание самих устройств. Поэтому информация об использовании символьных устройств дается только в общем контексте их применения в ОС.

4.2 BOOT-сектор и разделы винчестера

Учитывая тематику изучаемого материала, мы ограничимся рассмотрением конкретного физического устройства ЭВМ, известного как *винчестер* или «*жесткий диск*», который является *основным блочным устройством* для долговременного хранения информации и одновременно - *оперативно используется в работе ОС*.
Кроме того, именно на одном из разделов винчестера размещается файловая система, рассматриваемая ядром ОС как *корневая ФС*.

Все блочные устройства, на которых размещаются конкретные ФС имеют свою структуру, определяемую физическим типом устройства. Такую структуру имеют и

физические устройства, называемые *винчестер*. Широкое применение этого оборудования привело к стандартизации его общей структуры, которая уже была рассмотрена ранее в теме 2 (подраздел 2.6 «Винчестер и загрузочные устройства»):

- *классическая* структура MBR;
- *новая структура* GPT.

Широкое и успешное применение применение устройств типа винчестер и их общих структур привело к тому, что эти структуры стали переноситься на другие устройства, физически отличающиеся от винчестера. Примером таких устройств является flashUSB, которое является микросхемой и не имеет дисков, головок и секторов. Тем не менее, перенос структуры винчестера на flashUSB позволяет использовать это устройство как блочное, с наименьшими изменениями в системном ПО ОС.

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 поддерживамых 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.

Указанная структура демонстрируется схемой, представленной на рисунке 4.4.

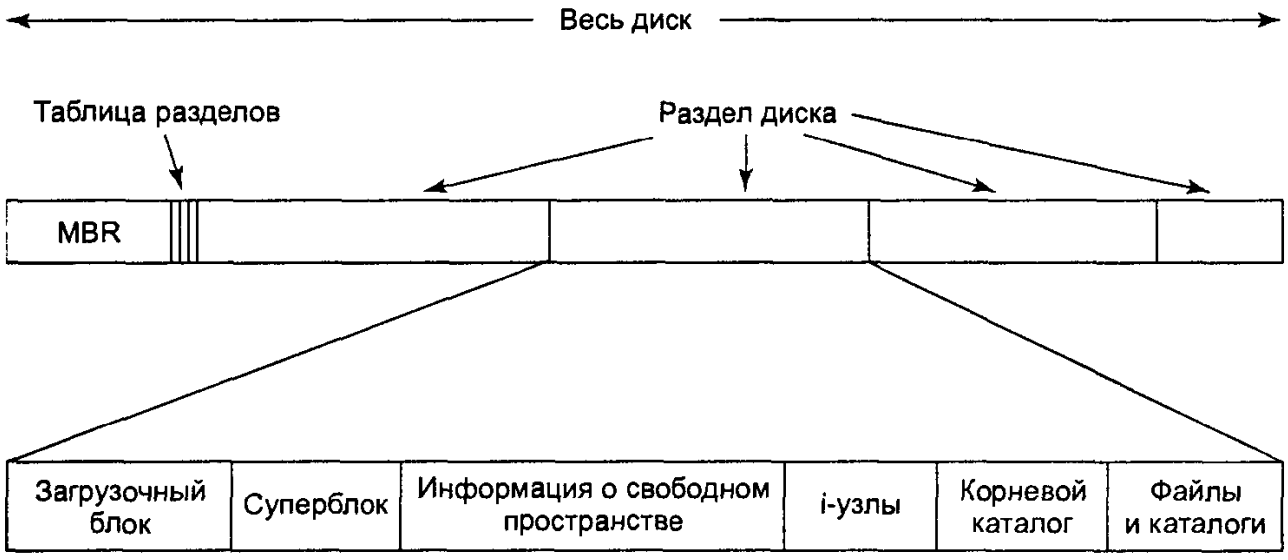

Рисунок 4.4 - Общая структура винчестера

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

Для примера, в ОС УПК АСУ, такими утилитами являются:

- **fdisk** - позволяет показать все блочные устройства ЭВМ, а также *обеспечить разбиение устройства* на нужное количество разделов;

- **mkfs** — позволяет *создать файловую систему на заданном разделе* заданного блочного устройства;
- **mknod —** позволяет *создавать узлы в ФС*, которые именуют блочные и символьные устройства для доступа к ним ПО пользовательского режима;
- **mount** — позволяет *монтировать (подключить) конкретную ФС* к древовидной структуре файлов ядра ОС;
- **umount** — позволяет *размонтировать (отключить) конкретную ФС* от древовидной структуры файлов ядра ОС.

Следуя основной парадигме UNIX-систем: «**Все есть файл**», разработчики ОС создавали *свои файловые системы*, которые *монтировались* при загрузке ОС и использовались, в дальнейшем, как *уникальные* хранилища информации.
Фактически, *задача загрузки первых реализаций ОС* сводилась (см. рисунок 4.4):

- *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.

Замечание

Многие корпорации такая ситуация вполне устраивала, но в 1985 году, компания Sun Microsystems для ОС SunOS предложила **VFS** — Virtual File System (File System Switch) - *Виртуальнаую Файловую Систему*.

4.3 Загрузочные сектора разделов

Для блочного устройства типа винчестер, минимальным физически читаемым объемом информации (данных) является сектор, который для современного оборудования ЭВМ равен **512 байт**.
Понятие сектора связано физической организацией обмена данными в блочных устройствах. Файловые системы (ФС), которые стали создаваться для различных нужд ОС, стали использовать *понятие блока*.
Понятие блока связано с логической организацией обмена данными между ОС и блочными устройствами. Соответственно размер блока может быть равен *одному сектору* или *кратен целому числу таких секторов*.

С другой стороны, понятие блока стали вводить с целью преодоления ограничений, которые накладываются способом адресации разделов в таблице MBR и ПО BIOS,

которые ограничивали возможный объем используемых блочных устройств. Например, в ОС MS Windows для форматирования блочных устройств стало использоваться понятие кластер.
Кластер — это последовательный набор секторов винчестера, который на логическом уровне рассматривается как единица обмена данными между ОС и блочным устройством.

С третьей стороны, 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). Выход из данной ситуации был найден с помощью LBA, которая саму адресацию секторов перекладывает на контроллер винчестера, а для ОС предоставляется упорядоченная последовательность логических блоков.
Наконец, разработка новой структуры блочных устройств GPT снимает проблему логической адресации болших объемов устройств, *расширив формат указателей адреса и стандартизировав размер LBA на уровне одного сектора (512 байт)*.

Другой круг проблем связан с ПО самой BIOS, рассмотренной нами в подразделах 2.2 и 2.3. Дело в том, что функции и прерывания BIOS могут работать только на уровне секторов и LBA, а также понимают структуру MBR. По этой причине, 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 это показано на рисунке 4.4. 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 стразу же после сектора MBR ПО загрузчика, что и используется нами при установке ПО GRUB.

Замечание

Использование альтернативных загрузчиков делает не нужным использование загрузочных секторов разделов, но, в силу преемственности структур ФС, данные сектора остаются.

Очевидно, что структуры различных ФС могут сильно отличаться друг от друга. На рисунке 4.4 приведена обобщенная структура ФС, характерная для ОС UNIX. Далее, на примере двух ФС — *FAT32* и *ext2fs*, мы рассморим их основные особенности, а затем проведем их сравнительный анализ.

4.4 Структура файловой системы FAT32 (VFAT)

FAT32 - файловая система компании *Microsoft*, разработанная ***в августе 1996 года***. Она является преемницей файловых систем ***FAT8***, ***FAT12*** и ***FAT16***, начало разработки которых положено Биллом Гейтсом и Марком МакДональдом, в ***1976 - 1977 годах***.
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 (для MS-DOS и MS Windows-9X – файл ***Io.sys***).

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 ***55AA*** (0xAA55).
BPB — *BIOS Parameter Block* — таблица содержашая множество параметров, определяющих характеристики блочного устройства и самой файловой системы.

Замечание

Для нужд FAT32, таблица BPB была расширена и названа ***BF_BPB***.
BF_BPB — *Big FAT BIOS Parameter Block*.
Размер загрузочного сектора:

- ***один*** физический сектор для FAT16;
- ***три*** физических сектора для FAT32.

Далее, в таблице 4.1, представлена структура первого сектора (***boot sector***) файловой системы FAT32.

Таблица 4.1 - Первый сектор раздела FAT32

Смещение, байт	***Длина поля, байт***	***Обозначение***	***Содержание***
0x00 (0)	3	*JUMP 90*	Безусловный переход на начало загрузчика
0x03 (3)	8		*Системный идентификатор* — начало *BF_BPB*
0x0B (11)	2	*SectSize*	Размер сектора, байт (обычно 512)
0x0D (13)	1	*ClastSize*	Число секторов в кластере
0x0E (14)	2	*ResSecs*	Число зарезервированных секторов, равно 32
0x10 (16)	1	*FATcnt*	Число копий FAT, обычно 2
0x11 (17)	2	*RootSize*	Количество элементов в корневом каталоге
0x13 (19)	2	*TotSect*	Общее число секторов. 0, если размер больше 32 Мб
0x15 (21)	1	*Media*	Дескриптор носителя
0x16 (22)	2	*FATsize*	Количество секторов на элемент таблицы FAT
0x18 (24)	2	*TrkSecs*	Число секторов на дорожке
0x1A (26)	2	*HeadCnt*	Число рабочих поверхностей (число головок)
0x1C (28)	4	*HidnSecs*	Число скрытых секторов, которые расположены перед загрузочным сектором. Используются при загрузке для вычисления абсолютного смещения корневого каталога и данных
0x20 (32)	4		Число секторов на логическом диске
0x24 (36)	4		Число секторов в таблице FAT
0x28 (37)	2		Расширенные флаги
0x2A (38)	2		Версия файловой системы
0x2C (39)	4		Номер кластера для первого кластера каталога. Обычно, значение этого поля равно 2.
0x34 (43)	2		Номер сектора с резервной копией загрузочного сектора
0x36 (54)	12		Зарезервировано FAT32 (расширение BF_BPB)
0x40 (64)	1		Физический номер устройства: 0x00 — флоппи диск; 0x80 — жесткий диск.
0x41	1	0	Зарезервировано для FAT32
0x42	1		Сигнатура расширенного загрузочного сектора.
0x43	4		Серийный номер тома. Случайное число, которое генерируется при фоматировании
0x47	11		Метка тома
0x52	8		Тип файловой системы (12-, 16-, 32-разрядная)

0x5A (90)	420		*Начало кода системного загрузчика*
0x1FE (510)	2		Сигнатура (слово AA55)

Поскольку содержимое загрузочной записи имеет для ОС важное значение, то *обычно имеются резервные копии этой записи*.
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*. В таблице 4.2, представлены возможные значения таблицы FAT.

Таблица 4.2 - Значения элементов таблицы FAT32 (используется 28 бит)

Значение	***Описание***
0x00000000	Свободный кластер
0x00000002 – 0x0FFFFFEF	Номер следующего кластера в цепочке
0x0FFFFFF0 – 0x0FFFFFF6	Зарезервированный кластер
0x0FFFFFF7	Плохой кластер
0x0FFFFFF8 – 0x0FFFFFFF	Последний кластер в цепочке

Замечание

В файловой системе FAT, между копиями загрузочной записи раздела и корневым каталогом, *находятся две копии таблиц FAT*.
При загрузке ОС таблица **FAT** загружается в оперативную память компьютера.
Для больших файловых систем таблица **FAT** может загружаться частями.

За таблицами FAT располагается *корневой каталог файловой системы* (***Root Directory***), размером:

512 байт, для **FAT16**.
2048 байт, для **FAT32**, сейчас — до 65535 элементов записей по 32 байта.

Корневой каталог файловой системы – список записей по ***32 байта***.
Исторически, сложилась ситуация, что файловая 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 вмещается в одну 32-байтовую запись.
Структура этого формата представлена в таблице 4.3.
Одинадцатый байт записи содержит поле «***Атрибуты***», значения которого представлены в таблице 4.4.

Для длинных имен файлов, в формате ***Unicode-16***, используется несколько 32-байтовых записей каталога в специальном формате. Один элемент, такого формата записи, показан на рисунке 4.5 и имеет несколько полей.

Таблица 4.3 - Основной вариант записи для короткого имени FAT32

Смещение			***Описание***
Hex	***Dec***	***Длина поля***	
00h	0	8 байт	Имя файла
08h	8	3 байт	Расширение файла
0Bh	11	1 байт	*Атрибуты файла*
0Ch	12	2 байт	Зарезервировано для NT
0Eh	14	2 байт	Время создания файла
10h	16	2 байт	Дата создания файла
12h	18	2 байт	Дата последнего доступа
14h	20	2 байт	*Старшее слово номера начального кластера*
16h	22	2 байт	Время последней записи
18h	24	2 байт	Дата последней записи
1Ah	26	2 байт	*Младшее слово начального кластера*
1Ch	28	4 байт	Размер файла в байтах

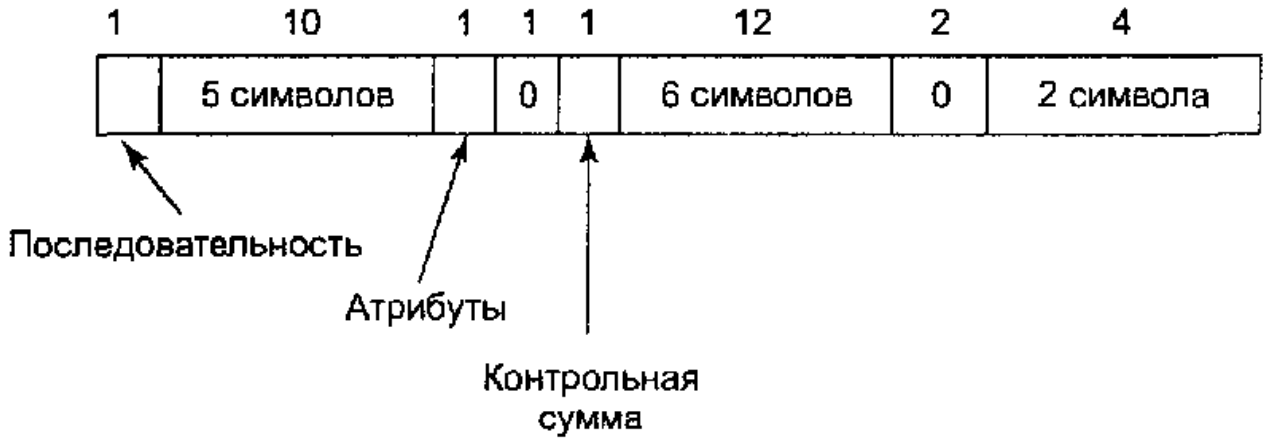

Рисунок 4.5 - Структура элемента записи для длинного имени файла

Таблица 7.4 - Значения 11-го байта атрибутов

7	***6***	***5***	***4***	***3***	***2***	***1***	***0***	***Hex***	***Значение***
0	0	0	0	0	0	0	1	01h	Только чтение
0	0	0	0	0	0	1	0	02h	Скрытый
0	0	0	0	0	1	0	0	04h	Системный
0	0	0	0	1	0	0	0	08h	Метка тома
0	0	0	1	0	0	0	0	10h	Подкаталог

0	0	1	0	0	0	0	0	20h	Архивный (измененный)
0	1	0	0	0	0	0	0	40h	Зарезервировано
1	0	0	0	0	0	0	0	80h	Зарезервировано

Область данных — все остальные кластеры раздела. Они используются для хранения *подкатологов* и *файлов*.
Подкатологи – 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.

Замечание

Чем больше размер кластера, тем больший размер раздела может адресоваться и повышается скорость обработки файлов, но уменьшается эффективность их хранения.

4.5 Структура файловой системы EXT2FS

ext2 (ext2fs) - *Second Extended File System* — базовая файловая система ОС Linux.
Разработана Реми Кардом и представлена *в январе 1993 года*.
Поддерживает файлы размером *до 2 Гбайт*.
Является преемницей ***ext*** - *Extended File System*, представленной *в апреле 1992 года*.

Структура дискового раздела ext2fs представляет *последовательность группы блоков*, которые нумеруются, начиная с 1.

Загрузочная запись – принадлежит 1-й группе блоков	Группа блоков 1	Группа блоков 2	...	Группа блоков n

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:

Супер-блок	Описание группы блоков (Group Descriptors)	Битовая карта блоков (Block Bitmap)	Битовая карта индексных дескрипторов (Inode Bitmap)	Таблица индексных дескрипторов (Inode Table)	Область блоков данных

Такая структура служит повышению производительности файловой системы:

- сокращается расстояние между таблицей индексных дескрипторов и блоками данных;
- сокращается время поиска нужного места головками в процессе операций записи/считывания файла.

Первый элемент группы блоков - *суперблок*, который одинаков для всех групп.
Все остальные элементы - индивидуальны для каждой группы.

Первые 1024 байта раздела занимает *загрузочная запись раздела*.
Суперблок, хранится в первой группе блоков и имеет ***смещение 1024 байта***.
Наличие нескольких копий суперблока объясняется чрезвычайной важностью этого элемента файловой системы.
Дубликаты суперблока используются при восстановлении файловой системы после сбоев.
Информация, хранимая в суперблоке (см. табл. 4.5), используется для организации доступа к остальным данным на диске. В суперблоке определяется размер файловой системы, максимальное число файлов в разделе, объем свободного пространства и содержится информация о том, где искать незанятые участки.

При запуске ОС, суперблок считывается в память и все изменения файловой системы вначале находят отображение в копии суперблока, находящейся в ОЗУ, и записываются на диск только периодически.
При выключении системы, суперблок обязательно должен быть записан на диск.

Таблица 4.5 - Структура суперблока

Название поля	***Тип***	***Комментарий***
s_inodes_count	ULONG	Число индексных дескрипторов в файловой системе
s_blocks_count	ULONG	Число блоков в файловой системе
s_r_blocks_count	ULONG	Число блоков, зарезервированных для *суперпользователя*
s_free_blocks_count	ULONG	Счетчик числа свободных блоков
s_free_inodes_count	ULONG	Счетчик числа свободных индексных дескрипторов
s_first_data_block	ULONG	Первый блок, который содержит данные. В зависимости от размера блока, это поле может быть равно 0 или 1.

s_log_block_size	ULONG	Индикатор размера логического блока: 0 = 1 Кб; 1 = 2 Кб; 2 = 4 Кб.
s_log_frag_size	LONG	Индикатор размера фрагментов
s_blocks_per_group	ULONG	Число блоков в каждой группе блоков
s_frags_per_group	ULONG	Число фрагментов в каждой группе блоков
s_inodes_per_group	ULONG	Число индексных дескрипторов (inodes) в каждой группе блоков
s_mtime	ULONG	Время, когда в последний раз была смонтирована файловая система.
s_wtime	ULONG	Время, когда в последний раз производилась запись в файловую систему
s_mnt_count	USHORT	*Счетчик числа монтирований* файловой системы. Если этот счетчик достигает значения, указанного в следующем поле (s_max_mnt_count), файловая система должна быть проверена (это делается при перезапуске), а счетчик обнуляется.
s_max_mnt_count	SHORT	Число, определяющее, сколько раз может быть смонтирована файловая система
s_magic	USHORT	"Магическое число" (***0xEF53***), указывающее, что файловая система принадлежит к типу ex2fs
s_state	USHORT	Флаги, указывающее текущее состояние файловой системы: является ли она чистой (clean) и т.п.
s_errors	USHORT	Флаги, задающие процедуры обработки сообщений об ошибках (что делать, если найдены ошибки).
s_pad	USHORT	Заполнение
s_lastcheck	ULONG	Время последней проверки файловой системы
s_checkinterval	ULONG	Максимальный период времени между проверками файловой системы
s_creator_os	ULONG	Указание на тип ОС, в которой создана файловая система
s_rev_level	ULONG	Версия (revision level) файловой системы.
s_reserved	ULONG[235]	Заполнение до 1024 байт (235 * 4 = 940)

Вслед за суперблоком расположено *описание группы блоков* (Group Descriptors). Это описание имеет *структуру длиной 32 байта*, представленную в таблице 4.6.

Таблица 4.6 - Структура описания группы блоков

Название поля	***Тип***	***Назначение***
bg_block_bitmap	ULONG	Адрес блока, содержащего битовую карту блоков (block bitmap) данной группы
bg_inode_bitmap	ULONG	Адрес блока, содержащего битовую карту индексных дескрипторов (inode bitmap) данной группы
bg_inode_table	ULONG	Адрес блока, содержащего таблицу индексных дескрипторов (inode table) данной группы
bg_free_blocks_count	USHORT	Счетчик числа свободных блоков в данной группе
bg_free_inodes_count	USHORT	Число свободных индексных дескрипторов в данной группе
bg_used_dirs_count	USHORT	Число индексных дескрипторов в данной группе, которые являются каталогами
bg_pad	USHORT	Заполнение
bg_reserved	ULONG[3]	Заполнение

Размер описания группы блоков можно вычислить как:

```
N = (размер_группы_блоков_в_ext2 * число_групп) / размер_блока
```

Информация, которая хранится в описании группы, используется, чтобы найти:

- битовые карты блоков,
- индексных дескрипторов,
- таблицу индексных дескрипторов.

Битовая карта блоков (***block bitmap***) - это структура, каждый бит которой показывает, отведен ли соответствующий ему блок какому-либо файлу.
Если бит равен 1, то блок занят.
Эта карта также служит для поиска свободных блоков в тех случаях, когда надо выделить место под файл.
Битовая карта блоков занимает число блоков, равное:

```
Bloks =  (число_блоков_в_группе / 8) / размер_блока
```

Битовая карта индексных дескрипторов выполняет аналогичную функцию по отношению к таблице индексных дескрипторов: показывает *какие именно дескрипторы заняты*.

Таблица индексных дескрипторов:
После битовой карты индексных дескрипторов, следует *таблица индексных дескрипторов файлов* - ***Inode Table***.

Каждая строка ***Inode Table*** имеет *размер 128 байт* и имеет *структуру индексного дескриптора файлов*, рассмотренного ниже. Количество строк ***Inode Table***, а следовательно, и размер таблицы, *задается при создании файловой системы*.

Замечание

Каждый файл требует одной записи в таблицу дескрипторов. Поэтому *размер этих таблиц нужно предусмотреть заранее*.

Среди индексных дескрипторов имеется *несколько значений дескрипторов*, которые *зарезервированы для специальных целей* и играют особую роль в файловой системе.

Замечание

Значения этих дескрипторов, которые перечислены в таблице 4.7, не могут быть использованы для других целей.

Таблица 4.7 - Специальные индексные дескрипторы

Идентификатор	***Знач.***	***Описание***
EXT2_BAD_INO	1	Индексный дескриптор, в котором перечислены адреса дефектных блоков на диске (Bad blocks inode).
EXT2_ROOT_INO	2	Индексный дескриптор корневого каталога файловой системы (Root inode).
EXT2_ACL_IDX_INO	3	ACL inode.
EXT2_ACL_DATA_INO	4	ACL inode.
EXT2_BOOT_LOADER_INO	5	Индексный дескриптор загрузчика (Boot loader inode).
EXT2_UNDEL_DIR_INO	6	Undelete directory inode.
EXT2_FIRST_INO	11	Первый незарезервированный индексный дескриптор.

Самый важный дескриптор в этом списке - *дескриптор корневого каталога*.
Каталог файловой системы - файл, состоящий из записей переменной длины.

Таблица 4.8 - Структура файла корневого каталога

Название поля	***Тип***	***Описание***
inode	ULONG	Номер индексного дескриптора (индекс) файла
rec_len	USHORT	Длина этой записи
name_len	USHORT	Длина имени файла
name	CHAR[0]	Первый символ имени файла

Замечание

Отдельная запись в каталоге не может пересекать границу блока, то есть должна быть расположена целиком внутри одного блока.
Если очередная запись каталога не помещается целиком в данном блоке, то она переносится в следующий блок, а предыдущая запись продолжается таким образом, чтобы она заполнила блок до конца.

Индексные дескрипторы файлов:
Вся информация о файле хранится в индексном десткрипторе (***Inode***).
Число файлов, которое может быть создано в файловой системе, *ограничено числом индексных дескрипторов*, которое:

- *либо явно задается*, при создании файловой системы;
- *либо вычисляется*, исходя из физического объема дискового раздела.

Размер индексного дескриптора = 32 * 4 = ***128 байт***.
Структура индексного дескриптора приведена в таблице 4.9.

Таблица 4.9 - Структура индексного дескриптора файла

Название поля	***Тип***	***Описание***
i_mode	USHORT	Тип и права доступа к данному файлу.
i_uid	USHORT	Идентификатор владельца файла (*Owner Uid*).
i_size	ULONG	Размер файла в байтах.
i_atime	ULONG	Время последнего обращения к файлу (Access time).
i_ctime	ULONG	Время создания файла.
i_mtime	ULONG	Время последней модификации файла.
i_dtime	ULONG	Время удаления файла.
i_gid	USHORT	Идентификатор группы (*GID*).
i_links_count	USHORT	Счетчик числа связей (Links count).
i_blocks	ULONG	Число блоков, занимаемых файлом.
i_flags	ULONG	Флаги файла (File flags)
i_reserved1	ULONG	Зарезервировано для ОС
i_block	ULONG[15]	Указатели на блоки данных, которые рассмотрены далее в разделе «Система адресации данных»
i_version	ULONG	Версия файла (для NFS)
i_file_acl	ULONG	ACL файла - *Access Control List – список доступа*
i_dir_acl	ULONG	ACL каталога - *Access Control List – список доступа*
i_faddr	ULONG	Адрес фрагмента (Fragment address)
i_frag	UCHAR	Номер фрагмента (Fragment number)

i_fsize	UCHAR	Размер фрагмента (Fragment size)
i_pad1	USHORT	Заполнение
i_reserved2	ULONG[2]	Зарезервировано

Среди множества полей индесного десткриптора файла, более подробно мы рассмотрим ***i_mode*** и ***i_block***.

Поле **i_mode** - тип и права доступа к файлу.
Оно является *двух-байтовым словом*, каждый бит которого служит *флагом*, индицирующим *отношение файла к определенному типу* или *установку одного конкретного права на файл* (таблица 4.10).

Таблица 4.10 - Поле прав доступа к файлу

Идентификатор	***Значение***	***Назначение флага (поля)***
S_IFMT	**F000**	**Маска для типа файла**
S_IFSOCK	A000	Доменное гнездо (socket)
S_IFLNK	C000	Символическая ссылка
S_IFREG	8000	Обычный (regular) файл
S_IFBLK	6000	Блок-ориентированное устройство
S_IFDIR	4000	Каталог
S_IFCHR	2000	Байт-ориентированное (символьное) устройство
S_IFIFO	1000	Именованный канал (fifo)
S_ISUID	0800	SUID - бит смены владельца
S_ISGID	0400	SGID - бит смены группы
S_ISVTX	0200	Бит сохранения задачи (sticky bit)
S_IRWXU	**01C0**	**Маска прав владельца файла**
S_IRUSR	0100	Право на чтение
S_IWUSR	0080	Право на запись
S_IXUSR	0040	Право на выполнение
S_IRWXG	**0038**	**Маска прав группы**
S_IRGRP	0020	Право на чтение
S_IWGRP	0010	Право на запись

S_IXGRP	0008	Право на выполнение
S_IRWXO	**0007**	**Маска прав остальных пользователей**
S_IROTH	0004	Право на чтение
S_IWOTH	0002	Право на запись
S_IXOTH	0001	Право на выполнение

Система адресации данных - одна из самых существенных составных частей файловой системы. Именно она позволяет находить нужный файл среди множества как пустых, так и занятых блоков на диске.
В ext2fs, она реализуется полем ***i_block*** индексного дескриптора файла.
Поле ***i_block*** в индексном дескрипторе файла представляет собой *массив из 15 адресов блоков*, что наглядно демонстрируется рисунком 4.6.

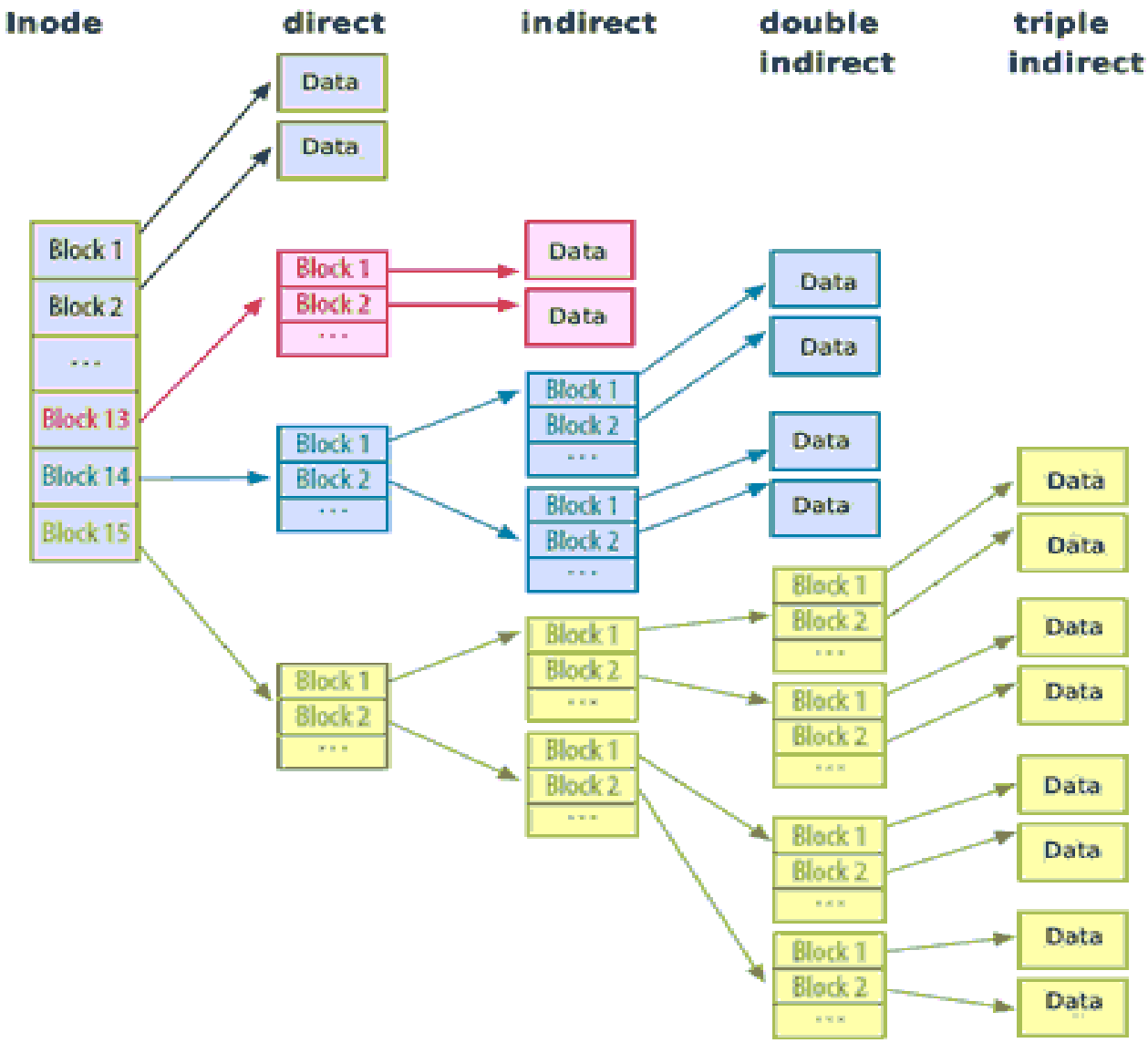

Рисунок 4.6 - Адресация блоков файла с помощью i_block

Первые 12 адресов в этом массиве **i_block[0 - 11]** = EXT2_NDIR_BLOCKS [0 - 11] - ***прямые ссылки*** (адреса) на номера блоков, в которых хранятся данные из файла.

Следующий адрес **i_block[12]** = EXT2_IND_BLOCK является ***косвенной ссылкой на блок***, в котором хранится ***список следующих адресов*** блоков с данными для этого файла:

```
Количество адресов =  (размер_блока / размер_ULONG)
```

Следующий адрес **i_block[13]** = EXT2_DIND_BLOCK указывает на ***блок двойной косвенной адресации*** (double indirect block). Этот блок содержит список адресов блоков, которые в свою очередь содержат списки адресов следующих блоков данных того файла, который задается данным индексным дескриптором:

$$\text{Количество адресов} = (\text{размер\_блока} / \text{размер\_ULONG})^2$$

Последний адрес **i_block[14]** = EXT2_TIND_BLOCK указывает на ***блок тройной косвенной адресации***:

$$\text{Количество адресов} = (\text{размер\_блока} / \text{размер\_ULONG})^3$$

Все оставшееся место, в группе блоков, отводится для хранения файлов.
Таким образом, прочитав суперблок, мы можем:

- определить адреса групп блоков данных прочитать групповые дескрипторы;
- из групповых дескрипторов определяем положение и размер индексных дескрипторов файлов;
- просматривая таблицу индексных дескрипторов файлов, мы определяем, что описывает дескриптор: файл, каталог, символьное или блочное устройство, другие объекты;
- по дескрипторам каталогов мы читаем блоки данных, в которых записаны имена файлов, каталогов, других устройств и соответствующие номера записей в таблице индексных дескрипторов файлов;
- номера свободных блоков и индексных дескрипторов файлов мы определяет по соответствующим битовым таблицам (картам).

4.6 Сравнение файловых систем

Рассмотренные выше структуры файловых систем ***FAT32*** и ***ext2fs*** демострируют два разных подхода к обработке данных:

- *простоту управления*, характерную для FAT32;
- *скорость обработки больших файлов*, характерную для ext2fs.

В любом случае, обе файловые системы имеют общую метаструктуру, которая определяется наличием:

- *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 подкатологи и файлы ФС.

Фактически, все известные файловые системы отличаются только *способом реализации этих элементов метаструктуры*.

Естественным образом, файловые системы ***FAT32*** и ***ext2fs*** имеют как свои преимущества, так и недостатки.
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 хранеия данных.

Именно эти недостатки потребовали от корпорации Microsoft, ***в марте 1993 года***, замены FAT32 на более сложную и современную ФС: ***ntfs***.

Файловая система ext2 берет свое начало от ФС ***minix***.
Файловая система ***ext*** - это ***Extended minix***.
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.

Указанные недостатки ext2 потребовали дальнейшее ее усовершенствование:

- **ext3** — *журналируемая ФС* появилась в ноябре 2001 года; поддерживает размер файлов до 16 ТБайт и размер раздела до 32 ТБайт;

- **ext4** — *журналируемая ФС* появилась в октябре 2008 года; поддерживает размер файлов до 16 ТБайт и размер раздела до 1 ЭБайт.

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*). Как только изменения файловой системы внесены в журнал, она применяет эти изменения к файлам или метаданным, а затем удаляет эти записи из журнала.
Записи журнала организованы в наборы связанных изменений файловой системы.

Замечание

Журналируемые записи первоначально появились в управлении базами данных, для которых характерно целостное управление большими массивами данных. С помощью их также реализуется *механизм транзакций*.

4.7 Стандартизация структуры ФС

Предыдущий учебный материал дает описание блочного устройства на уровне:
- *секторов, LBA и разделов* винчестера: подразделы 4.1 и 4.2;
- *специализированных блоков отдельных ФС*, размещаемых в отдельном разделе винчестера: подразделы 4.3 — 4.6;

Данный учебный материал рассматривает структуру ФС как организованый 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.

На рисунке 4.7 приведен пример дерева каталогов и файлов некоторой ФС типа UNIX.

Хорошо видно, что это дерево ФС логически структурировано в прикладном плане, например:
- *dev* — директория размещения узлов устройств ЭВМ (devices);
- *usr* — директория ориентированная на пользователей (users);
- *home* — директория для рабочих (домашних) областей пользователей;
- *bin* — директория расположения исполняемых файлов ОС (binary).

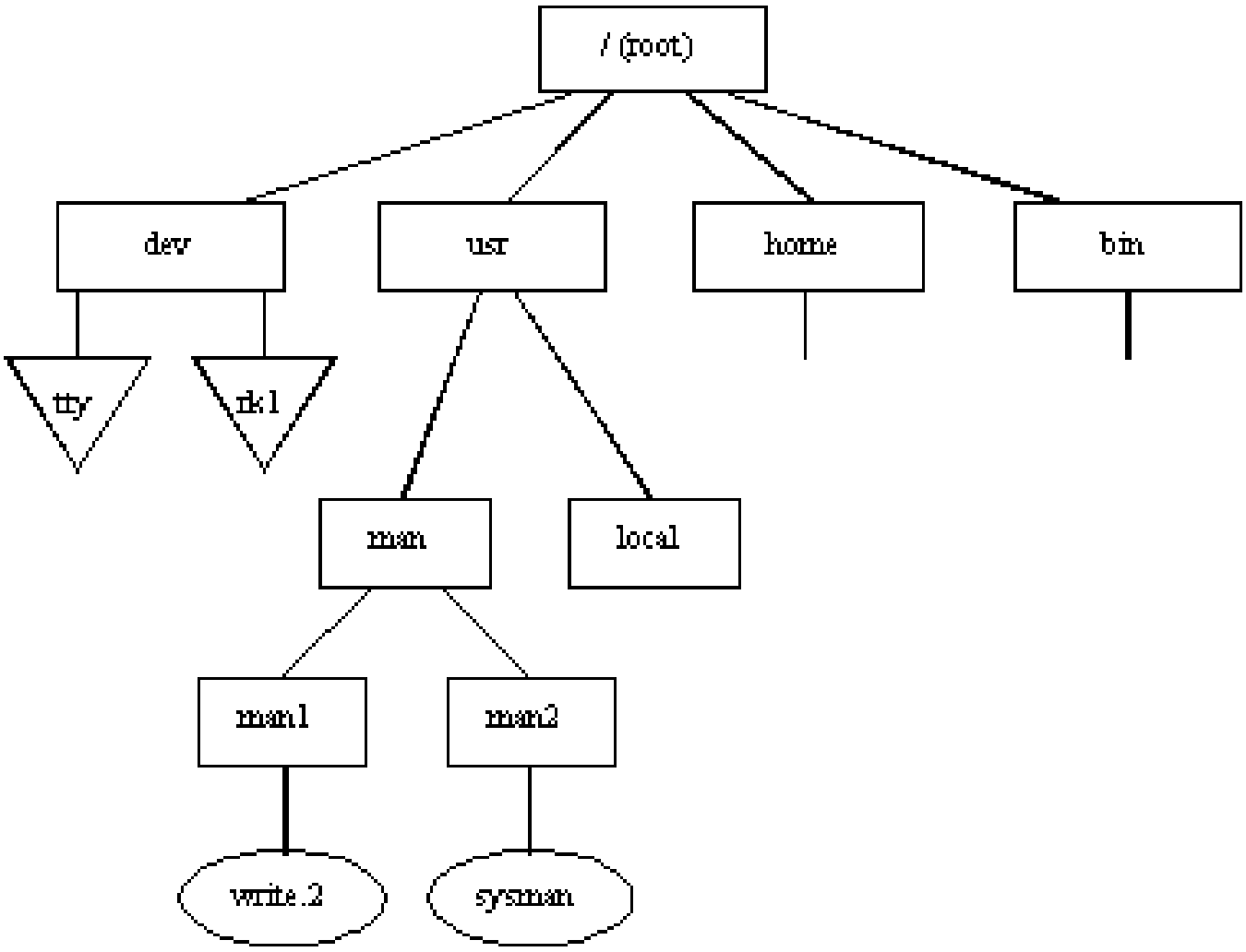

Рисунок 4.7 - Пример структуры ФС типа UNIX

В августе 1993 года, в рамках проекта GNU/Linux, стал разрабатываться стандарт на структуру файловой системы (*Filesystem Standard*), который был выпущен *в феврале 1994 года* и получил обозначение ***FSSTND***.
Позже, *в 1996 году*, была организована группа *Free Standard Group* (***FSG***), к которой присоединились и другие разработчики ОС.

FSG продолжила разработку структуры иерархии ФС, которая была бы пригодна для всех UNIX-подобных систем. Такой стандарт получил название ***Filesystem Hierarchy Standard*** (**FHS**).

В общем случае, файловая система Linux разделена на *три крупных уровня иерархий*.
Кратко рассмотрим каждый из этих уровней.

Первый уровень иерархии ФС составляют каталоги, которые управляются пользователем ***root*** и другими *администраторами системы*.

/	*Корневой каталог*. Должен содержать все, что необходимо для стар-та и запуска ОС, а также на случай восстановления ее из резервной копии.
/bin	Необходимые программы (бинарные файлы).
/boot	Файлы загрузчика LILO (GRUB)
/dev	Устройства (все устройства доступны в виде файлов).
/etc	Файлы конфигурации.
/etc/X11	Файлы конфигурации для X Window.
/home	Домашние каталоги пользователей.
/home/reznik	Домашний каталог пользователя reznik.
/lib	Разделяемые библиотеки, необходимые для работы программ, находящиеся в каталогах /bin, /sbin и др.
/lib/modules	Модули ядра.
/media	Директория монтирования ***временных ФС*** (floppy и т.д)
/mnt	Точка для временного монтирования файловых систем.
/opt	Каталог для пакетов программ.
/opt/Office50	Пакет Star Office 5.0 (в данном случае)
/proc	*Специальные виртуальные файлы*, обеспечивающие доступ к функциям ядра Linux.
/root	Домашний каталог пользователя root.
/sbin	Необходимые системные программы (бинарные файлы).
/tmp	*Временные файлы* для всех программ. Эти файлы могут уничтожаться системой при каждой перезагрузке или периодически.
/usr	***Второй уровень иерархии***.
/var	Изменяемые данные.

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.

/usr/X11R6	Система X Window, версия 11.6
/usr/X386	Система X Window, версия 11.5
/usr/bin	Программы, предназначенные для пользователей.
/usr/games	Игры и развлечения.
/usr/include	Файлы заголовков для программ на языке C.

/usr/lib	Библиотеки.
/usr/local	Локальная иерархия (***корень третьего уровня иерархии***). Обычно сюда монтируются всякие дополнения.
/usr/sbin	Системные программы.
/usr/share	**Архитектурно независимые данные**:
/usr/share/dict	Словари.
/usr/share/doc	Документация о свободных форматах.
/usr/share/games	Игры.
/usr/share/info	Документация в формате GNU info.
/usr/share/locale	Данные для системной локали.
/usr/share/man	Документация в формате **man**.
/usr/share/nls	Данные для поддержки ***национальных языков*** (NLS).
/usr/share/misc	Все остальные данные.
/usr/share/terminfo	База данных terminfo.
/usr/share/tmac	Макросы для troff.
/usr/share/zoneinfo	База данных и ***конфигурация местного времени***.
/usr/src	Исходные тексты.
/usr/src/linux	Исходные тексты ядра Linux.

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

Далее, более подробно рассмотрим корневую директорию изменяемых данных ***/var***, поскольку она содкржит множество системных данных ОС.

/var	*Каталог*, содержимое которого может меняться при работе системных программ.
/var/cache	Временно сохраняемые файлы различных программ (браузеров Internet, программы man, сервера шрифтов и другие).
/var/games	Изменяемые файлы для игр из /usr/games.
/var/lock	Lock-файлы, указывающие, что то или иное устройство (обычно модем) занято.
/var/log	Системные журнальные файлы.
/var/mail	Почтовые ящики пользователей.
/var/opt	Изменяемые файлы для программ из **/opt**.
/var/run	Файлы, имеющие отношение к запущенным в настоящий момент программам, например, хранящие номера процессов.

/var/spool	*Данные, обработка которых отложена* (статьи из групп новостей, документы посланные на печать, письма, которые не удалось сразу послать из-за проблем со связью и другие).
/var/state	Данные, имеющие отношение к состоянию программ, напри-мер, **backup**-файлы, файлы текстовых редакторов, конфигура-ции, имеющие отношение ко всей системе и прочее.
/var/tmp	Временные файлы, которые не должны регулярно уничтожаться системой.
/var/yp	Файлы системы ***NIS*** (Network Information Services).

Замечание

Видимое пользователям дерево ФС отличается от дерева ФС доступного ядру ОС:

- ядро ОС «видит» дерево виртуальной ФС (**VFS**);
- пользователь «видит» только ветвь дерева, объявленного для пользователя как корень **root**;
- современная тенденция контейнерная технология склоняется к минимизации видимой части ФС, повышая ее защищенность.

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:

- чтобы определить, где находится пользователь, следует использовать команду ***pwd***, которая в качестве результата возвратит имя директории;
- чтобы перемещаться по ФС, пользователь должен использовать команду:

```
cd [директория],
```

где аргумент команды обрабатывается по следующим правилам:

- отсутствие аргумента *делает текущей директорию*, заданную системной переменной **HOME**;
- аргумент, начинающийся символом /, *задает абсолютные координаты директории*, начиная от корня ФС;
- аргумент, начинающийся любым допустимым символом, отличным от /, задает *относительные координаты директории*, начиная от текущей директории;
- *для уточнения* использования относительной адресации, можно использовать аргумент ***./директория***;
- *для относительной адресации* от родительской директории, можно использовать аргумент ***../директория***

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.

Например,

```
$  ls -l
итого 864
drwxrwxrwx     5 root root   4096 окт.  6   2013  asu11upk3
drwxrwxrwx     3 root root   4096 июля  28  2013  asu12upk
-rw-rw-r--     1 vgr  vgr    18   дек.  22  2012  asu_tmp
drwxrwxr-x     2 vgr  vgr    4096 июня  1   02:32 bin
drwxrwxr-x     2 vgr  vgr    4096 дек.  26  2012  bin.old
-rw-r--r--     1 vgr  vgr    5860 янв.  5   2013  casper.log
-rw-rw-r--     1 vgr  vgr    180  сент. 18  2012  demo2.txt
-rw-rw-r--     1 vgr  vgr    490  сент. 18  2012  demo.txt
…
$
```

Кроме специального назначения директории **/dev**, о которой говорилось ранее, ФС имеет директорию **/proc**, в которую монтирована файловая система **proc**. В ней содержатся множество файлов, содержащих информацию о параметрах или функциональных возможностях ядра ОС.

Например, чтобы узнать, какие типы ФС поддерживает ядро ОС, можно выполнить:

```
$ cat /proc/filesystems
nodev   sysfs
nodev   rootfs
nodev   bdev
nodev   proc
nodev   cgroup
nodev   cpuset
nodev   tmpfs
nodev   devtmpfs
nodev   debugfs
nodev   securityfs
nodev   sockfs
nodev   pipefs
nodev   anon_inodefs
nodev   devpts
        ext3
        ext4
nodev   ramfs
nodev   hugetlbfs
```

```
            vfat
nodev       ecryptfs
            fuseblk
nodev       fuse
nodev       fusectl
nodev       pstore
nodev       mqueue
$
```

Данный пример хорошо показывает, что *не все типы ФС* подразумевают наличие «физических» (*блочных*) устройств.

Другой пример, - файл */proc/cmdline*, который содержит список параметров, переданных ядру во время загрузки ОС:

```
$  cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.5.0-21-generic root=/dev/sda5 ro
$
```

Замечание

Последний пример - чисто демонстрационный. Студенту самостоятельно следует определить и описать содержимое файла /proc/cmdline.

4.8 Модули и драйверы ОС

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

Чтобы работать с конкретным устройством, ядро ОС имеет специальное ПО, которое называется *драйвером*.
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.

Поскольку без некоторой части драйверов ОС не сможет даже загрузиться, такие драйверы включаются в ядро ОС ***статически***. Обычно, перед компиляцией ядра ОС, запускается **программа-конфигуратор**, с помощью которой определяется: какие драйверы будут включаются в ядро ***статически***, а какие будут подгружаться ***динамически:*** во время загрузки ОС или по мере необходимости.

Идея использования модулей ядра — очень привлекательна:

- *для разработки любого ПО* используются средства разработки, которые *сами являются ПО* и могут функционировать *только в среде уже загруженной ОС*;
- *статически скомпилированный драйвер* может вступить в конфликт с архитектурой аппаратной части ЭВМ, что может сделать ядро ОС *неработоспособным*;
- *стремление уменьшить размер ядра ОС*, которое является актуальным для *встроенных (embedded) систем* или в *проектах микроядерной архитектуры ОС*.

Для создания модулей в ОС Linux разработана специальная технология, которая широко освещена в соответствующей литературе. Например, корпорация IBM на сайте http://www.ibm.com/developerworks/ru/library/l-linux_kernel_01/ выпустила более 70-ти статей, посвященных этой тематике.

В общем случае, **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*.

4.9 Системные вызовы ОС по управлению устройствами и ФС

Один из общепринятых способов повышения мобильности ПО - обеспечение стандартизации окружения приложений. Это подразумевает стандартизацию *ПО ОС*, *утилит* и *программных интерфейсов* саих приложений. Таким средством является

стандарт **POSIX** (*Portable Operating System Interface*), который описывает различные интерфейсы операционной системы.

Название POSIX предложено известным специалистом, основателем Фонда свободного программного обеспечения, Ричардом Столмэном. Наиболее современная версия стандарта POSIX, ***в редакции 2003 г.***, основана на Техническом стандарте *Open Group IEEE Std 1003.1* и на международном стандарте *ISO/IEC 9945*.

По состоянию на 2001 год, стандарт содержал следующие *четыре части*:

1. основные определения (термины, концепции и интерфейсы, общие для всех частей);
2. описание прикладного программного С-интерфейса к системным сервисам;
3. описание интерфейса к системным сервисам на уровне командного языка и служебных программ ;
4. детальное разъяснение положений стандарта, обоснование принятых решений.

В дальнейшем, накапливались и были внесены многие мелкие исправления, учтенные *в редакции 2003 года*.

Стандарт POSIX - обязательный элемент современной дисциплины разработки прикладных систем.

Таким образом, рассмотрев основные понятия и требования стандарта POSIX, мы можем перейти к непосредственному обсуждению системных вызовов ОС. На рисунке 1.9, приведен список системных вызовов ОС, разделенных на группы:

- управление процессами;
- управление файлами;
- управление каталогами и файловыми системами;
- разные, которые нельзя полностью отнести к предыдущим группам.

Непосредственно к нашей теме, относятся группы *управление файлами*, а также *управление каталогами и файловыми системами*.

Примеры предыдущего раздела показывают, что зная структуру необходимых данных, можно одни функции реализовать через другие, «более мелкие», но общая парадигма системного ПО требует отношение к ним как к функциям одного уровня.

Замечание.

Непосредственное применение системных вызовов предполагает изучение правил их вызова с использованием утилиты ***man***, которая, собственно говоря, и была создана для этих целей.

Наряду с низкоуровневыми системными вызовами, для работы с файлами и файловыми системами, используются более высокоуровневые функции, которые обеспечивают буферизованный ввод-вывод и подробно изложены в любом учебнике по языку С.

Вызов	Описание
Управление процессами	
Pid=fork()	Создает дочерний процесс, идентичный родительскому
Pid=waitpid(pid, &statioc, options)	Ожидает завершения дочернего процесса
s=execve(name, argv, environp)	Перемещает образ памяти процесса
Exit(status)	Завершает выполнение процесса и возвращает статус
Управление файлами	
fd=open(file, how, ...)	Открывает файл для чтения, записи
s=close(fd)	Закрывает открытый файл
n=read(fd, buffer, nbytes)	Читает данные из файла в буфер
n=write(fd, buffer, nbytes)	Пишет данные из буфера в файл
Position=lseek(fd, offset, whence)	Передвигает указатель файла
s=stat(name, &buf)	Получает информацию о состоянии файла
Управление каталогами и файловой системой	
s=mkdir(name, mode)	Создает новый каталог
s=rmdir(name)	Удаляет пустой каталог
s=link(name1, name2)	Создает новый элемент с именем
s=unlink(name)	Удаляет элемент каталога
s=mount(special, name, flag)	Монтирует файловую систему
s=umount(special)	Демонтирует файловую систему
Разные	
s=chdir(dimame)	Изменяет рабочий каталог
s=chmod(name, mode)	Изменяет биты защиты файла
s=kill(pid, signal)	Посылает сигнал процессу
Seconds=time(&seconds)	Получает время, прошедшее с 1 января 1970 г.

Рисунок 7.9 - Системные вызовы стандарта POSIX

4.10 Три концепции работы с устройствами

Для того чтобы, чтобы уже созданные ФС стали доступны приложениям ОС, *должны быть созданы узлы* (*node*), которые расположены в директории /dev.
Узлы ОС могут быть созданы из пространства пользователя:

- *с помощью системного вызова* ***mknod(...)*** из программы на языке ***C***;
- *с помощью утилиты* ***mknod*** посредством команды языка ***shell***.

Хотя любой из указанных способов предполагает указание всего четырех параметров - ***имя_узла***, ***тип_узла***, ***старший_номер_узла*** и ***младший_номер_узла***, такая ситуация создает множество проблем:

- *какое имя дать узлу*, чтобы оно мнемонически отображало *тип устройства* и было не очень длинным, для удобства работы с ним;
- *какой старший номер* присвоить устройству, чтобы он соответствовал типу устройства и *был стандартным для этого типа устройств*;
- *сколько должно быть младших номеров устройства*, чтобы удовлетворить все варианты использования ОС;

- *что делать с узлами устройств*, которых нет в архитектуре ЭВМ;
- *как узнать момент*, когда к ЭВМ устройство подключается динамически (*временное подключение и отключение устройств*).

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

Концепция manual (ручная) появилась первой. Она предполагает использование, упомянутых выше, 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**;
- *винчестеры, с контроллерами* ***ATA***, стали именоваться **/dev/hda**, **/dev/hdb** и далее, а разделы, на которых фактически и создаются ФС - **/dev/hda1**, **/dev/hda2**, …, **/dev/hdb1**, **/dev/hdb2**, …
- *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**.

Замечание

Используя скрипт **MAKEDEV**, можно легко создать узлы с корректными именами и параметрами, но невозможно решить проблемы эффективного их использования. Особенно остро этот вопрос встал, когда корпорация Microsoft стала интенсивно призвать производителей оборудования внедрять *технологию Plug and Play*. Принципиальный аспект этой проблемы состоит в том, что драйвера устройств работают в пространстве ядра ОС. И хотя они могут определить факт подключения к ЭВМ нового устройства, *они не предназначены для посылки сигналов процессам*, работающим в режиме пользователя.

Концепция udev, реализованная ***в ноябре 2003 года***, решила проблему внедрения *технологии Plug and Play*.
Здесь следует отметить, что широкое внедрение графических оболочек ОС потребовало организовать *эффективный обмен сообщениями между демонами (фоновыми процессами) рабочего стола*. Для этого была разработана *технология D-Bus* - специальная система межпроцессного взаимодействия, которая предоставляет *несколько шин (логических шин)*:

- *системную шину*, которая создается *при старте демона D-Bus* и обеспечивает взаимодействие демонов системных процессов;
- *сессионную шину*, которая создается для каждого пользователя, авторизовавшегося в системе, и обеспечивает взаимодействие его личных процессов.

Таким образом, 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 неиспользованных узлов*.

Концепция systemd стала внедряться ***с апреля 2010 года*** и еще не реализована многими ОС.
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*).**

ОС УПК АСУ, для работы с узлами устройств, использует эту современную концепцию systemd.

4.11 Разделы дисков и работа с ними

Когда все ФС, на всех разделах блочных устройств созданы, работа с файловой системой ОС становится достаточно простой:

- *необходимо определить точки монтирования* (директори) корневой ФС ОС для подключения к ним (монтирования) нужных сопуствующих 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.

Хотя такой подход требует определенной квалификации пользователя и ориентирован больше на администраторов серверов, он вполне приемлем для многих случаев, когда аппаратная конфигурация ЭВМ является стабильной и не требует серьезных обновлений.
Ситуация обостряется и становится критичной, когда необходимо провести модернизацию свзанную с:

- *динамическим подключением других* («не родных») ФС;
- *подключением к сетевым ресурсам данных*, которые кроме проблем самой сети вынуждают использовать ФС, построенные на иных концепциях, например, ФС типа FAT32 и ntfs, которые не используют атрибутов пользователя, группы и других.

В общем случае, эти проблемы решаются разработкой и использованием необходимых модулей ядра ОС, что уже было рассмотрено в подразделе 7.8. 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.

В 1985 году, компания Sun Microsystems для ОС SunOS предложила ***VFS*** — Virtual File System (File System Switch) - *Виртуальнаую Файловую Систему*.

Виртуальная файловая система VFS:
Суть идеи — расположить VFS между *приложениями* и *конкретными файловыми системами* на внешних носителях. Это позволяет пользовательского интерфеса для ФС *разных типов* сконцентрировать в одном месте, что *создает утифицированный эталон для разработки драйверов ОС*.
Таким образом, **VFS** обеспечивает *унифицированный программный интерфейс* к услугам файловой системы, причем безотносительно к тому, какой тип файловой системы (vfat, ext2fs, nfs и другие) имеется на конкретном физическом носителе.
В результате, 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.

В любом случае, возможность применения тех или иных типов ФС зависит как от потребности самих вычислительных систем, реализованных на ПО ОС, так и от возможности правильного управления имеющимися ФС. Все равно, основу уравления ФС составляют утилиты ***mount*** и ***umount***, которые мы рассмотрим в следующем подразделе.

4.12 Монтирование и демонтирование устройств

Монтирование файловой системы — *подключение конкретного раздела* внешнего блочного устройства *к конкретному каталогу ФС*, в видимой пользователю общей части дерева корневой ФС.
Дентирование файловой системы — *отключение конкретного раздела* внешнего блочного устройства от *конкретного каталога ФС*, в видимой пользователю общей части дерева корневой 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* ***umount*** с указанием или узла ФС или директории монтирования, при этом: *старое содержимое директории становится видимым, а демонтируемая ФС — невидимой*;

Простейший общий вид команды монтирования:

```
mount -t Тип Устройство Директория;
```

Например, монтирование устройства ***/dev/sda2***, типа ***ext2*** в каталог ***/mnt***, выглядит так:

```
$  mount -t ext2 /dev/sda2 /mnt
$
```

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

```
/dev/sda5    on /       type ext4 (rw,errors=remount-ro)
proc         on /proc   type proc (rw,noexec,nosuid,nodev)
sysfs        on /sys    type sysfs (rw,noexec,nosuid,nodev)
none         on /sys/fs/fuse/connections type fusectl (rw)
…....
/dev/sda2    on /mnt    type ext2 (rw)
```

*Команда **umount**, для указанного примера*, может быть выполнена в двух вариантах:

```
    umount /dev/sda2;
    umount /mnt;
```

Замечание

Информация о действиях команды mount оперативно заносится в файл **/etc/mtab** в формате:

```
    Устройство on Каталог type Тип (Опции)
```

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.

Демонстрационный пример **/etc/fstab**:

```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/sda2 during installation
/dev/sda4   /               ext2    errors=remount-ro        0       1
# swap was on /dev/sda6 during installation
/dev/sda6   none            swap    sw                       0       0
/dev/fd0    /media/floppy0  auto    rw,user,noauto,exec,utf8 0       0
```

Замечание

Исторически, ***опции***, с которыми монтируются ФС, создавались разработчиками разных ФС.

Такая ситуация требует отдельного изучения опций для каждой ФС. Кроме того, ***/etc/fstab*** имеет свои специфические опции.

Набор наиболее распространенных опций:

async	Весь ввод/вывод осуществляется асинхронно
sync	Весь ввод/вывод осуществляется синхронно
atime	Обновлять времена обращения (по умолчанию)
noatime	Времена обращения не обновляются
auto	Система ***может*** быть смонтирована с опцией **-a (для всех)**
noauto	Система ***не может*** быть смонтирована с опцией **-a (для всех)**
defaults	Аналог ***rw,suid,dev,exec,auto,nouser,asunc***
dev	Файлы устройств ФС интерпретируются как устройства
nodev	Файлы устройств запрещены
exec	Право на запуск исполняемых файлов
noexec	Запрет на запуск исполняемых файлов
suid	Биты SUID и SGID ***действуют***
nosuid	Биты SUID и SGID **не *действуют***
user	Обычный пользователь может смонтировать систему. Подразумевает опции: ***noexec,nosuid,nodev***
nouser	Только **root** может монтировать систему
remount	Перемонтировать уже смонтированную систему (например, для изменения опций системы)
ro	Смонтировать ФС только для чтения
rw	Смонтировать ФС для чтения и записи

4.13 Файловые системы loopback, squashfs, overlayfs и fuse

Наряду с рассмотренными выше ФС на разделах дисков и в ядре ОС, используется множество специальных ФС. Мы рассмотрим четыре из них: ***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**. Следующий пример показывает вариант создания десяти циклических устройств: ***/dev/loop1***, …,

/dev/loop9.

```
    for x in 0 1 2 3 4 5 6 7 8 9; do
        [ -e /dev/loop$x ]  ||  mknod /dev/loop$x  b  5 $x
    done
```

Циклические ФС обычно создаются в файлах, которые могут быть *сжатыми* и *зашифрованными*. Для примера рассмотрим создание ФС типа ***ext2fs*** в файле */tmp/ramdisk* и последующее монтирование созданной ФС в директорию */mnt/initrd*:

```
# Используемые константы
BLKSIZE=1024 # Размер блока в байтах
RDSIZE=4000  # Количество блоков в ФС
RBLOCK=0     # Число резервных блоков

# Создаем файл, забитый нулями
# (dd – преобразование и копирование файлов)
dd if=/dev/zero of=/tmp/ramdisk bs=$BLKSIZE count=$RDSIZE

# Создаем ФС в файле ext2fs
/sbin/mke2fs -F -m $RBLOCK -b $BLKSIZE /tmp/ramdisk $RDSIZE

# Монтируем ФС в файле к директории через устройство /dev/loop0
mount /tmp/ramdisk /mnt/initrd -t ext2 -o loop=/dev/loop0
```

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 */asu14.04upk/casper/filesystem.squashfs*, то монтирование такой ФС (типа *squashfs*) к директории */rofs* ядра ОС осуществляется двумя командами:

```
# Монтируем устройство /dev/sda1 в директорию /cdrom
mount /dev/sda1 /cdrom -t ntfs -o ro

# Монтируем ФС в файле к директории /rofs через устройство /dev/loop1
mount /cdrom/asu14.04.upk/casper/filesystem.squashfs /rofs \
     -t squashfs -o loop=/dev/loop1
```

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*.
Работы над КФС ведутся ***с середины 90-х годов***. ***В январе 2007 года***, в некоторые дистрибутивы Linux, была включена КФС, названная *UnionFS*. Альтернативная разработка *Aufs (AnotherUnionFS)* ведется ***с 2006 года***.

Каскадно-объединенное монтирование — *одновременное монтирование разных нескольких файловых систем как одну*, например для объединения ФС нескольких сайтов.
Overlayfs обеспечивает надежное монтирование двух файловых систем на разных уровнях. Рассмотрим подробнее этот вопрос на примере:

- пусть устройство CDROM монтировано в директорию **/cdrom** с возможностью ***только чтение***;
- пусть в директорию **/mnt/initrd** монтирован, созданный ранее ***ramdisk***, который имеет возможность ***чтения и записи***;
- проведем каскадно-объединенное монтирование в директорию **/tmp/mnt**, предполагая, что ***CDROM*** будет монтирован на нижнем уровне, а ***ramdisk*** — на верхнем;
- учтем, что в ядре ОС имеется имя **overlayfs**, которое не является устройством, но позволяет монтировать ФС.

Тогда, команда монтирования будет иметь вид:

```
# Команда каскадно-объединенного монтирования
mount -t overlay overlay /tmp/mnt -o  \
    rw,upperdir=/mnt/initrd,lowerdir=/cdrom
```

В результате такого действия, в директории /tmp/mnt, мы будем видеть все файлы директории /cdrom. Кроме того, мы можем редактировать и использовать любой файл директории /cdrom, но сохраняться эти файлы будут в памяти *ramdisk*.
Последня специальная ФС, которую мы кратко рассмотрим — ***FUSE***.
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.
На рисунке 4.8 показана схема взаимодействия системных вызовов пользователей с ядром ОС и ПО в пространстве пользователя.
Хорошо видно, что:

- *схема взаимодействия* системного и прикладного ПО напоминает аналогичное взаимодействие ***микроядра*** и ***серверных сервисов***;
- *кроме системной библиотеки* ***glibc*** необходима библиотека ***libfuse***.

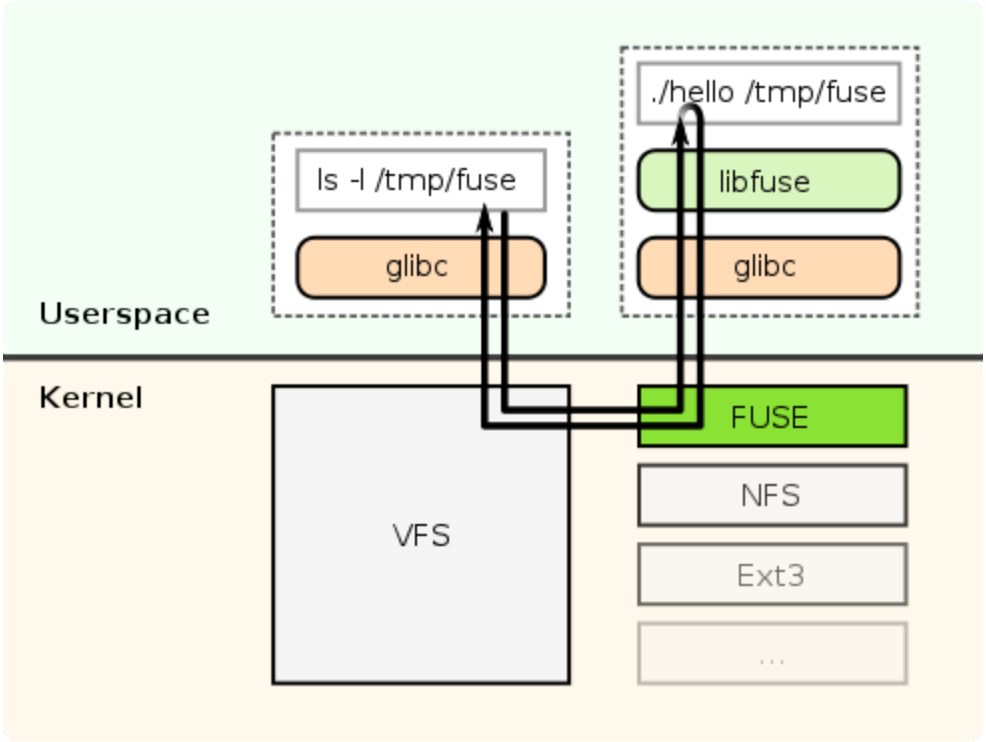

Рисунок 4.8 - Схема работы модуля FUSE

Для операций монтирования и демонтирования используется специальная утилита ***fusermount***, использующая файл конфигурации */etc/fuse.conf*:

```
fusermount [OPTIONS] MOUNTPOINT
```

Опция ***-u*** используется для демонтирования FUSE.

Следует заметить, что многие видные разработчики ОС критикуют FUSE, но эти вопросы выходят за рамки нашей дисциплины.

Замечание

Разделы винчестера, содержащие ФС ***ntfs***, монтируются в Linux с типом ***fuseblk***.

4.14 Дисковые квоты

В заключение темы, рассмотрим вопросы ограничения размеров, которые системно можно накладывать на файловые системы.
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***:

```
$  df
Файл.система   1K-блоков Использовано Доступно Использовано% Смонтировано в

/dev/sda5       21356772     14841812  5430084           74% /
udev             1669124           12  1669112            1% /dev
tmpfs             670748          780   669968            1% /run
none                5120            0     5120            0% /run/lock
none             1676860           84  1676776            1% /run/shm
none              102400           12   102388            1% /run/user
/dev/sda1       51199120     26771180 24427940           53% /cdrom
$
```

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.

4.15 Лабораторная работа по теме №4

Цель лабораторной работы №4 — практическое закрепление учебного материала по теме «Управление файловыми системами ОС».
Метод достижения указанной цели — закрепление учебного материала, изложенного в первом разделе пособия посредством утилит ОС, а также выполнение заданий, приведенный в данном разделе.
Чтобы успешно выполнить данную работу, студенту следует:

- *запустить с flashUSB* ОС УПК АСУ, подключить личный архив и переключиться в сеанс пользователя ***upk***;
- *запустить на чтение* данное пособие и на редактирование личный отчет;
- *открыть одно или несколько окон терминалов*, причем хотя бы в одном окне терминала открыть Midnight Commander, для удобства работы с файловой системой ОС;
- *приступить к выполнению работы*, последовательно пользуясь рекомендациями представленных ниже подразделов.

Замечание

Многие команды ОС студенту еще не известны, поэтому следует:

- для вывода на консоль руководства по интересуемой команде, использовать: ***man имя_команды***;
- для выяснения сущесвования команды, ее доступности и местоположения, использовать: ***command -v имя_команды***;
- для уточнения правил запуска конкретной команды, можно попробовать один из вариантов: ***команда --help*** или ***команда -h*** или ***команда -?***.

В процессе выполнения лабораторной работы студент заполняет личный отчет по каждому изученному вопросу!

4.15.1 Типы, имена и узлы устройств

Прочитайте и усвойте материал подраздела 4.1.
Исследуйте содержание директории */dev*.
Исследуйте содержание директории */proc*.
С помощью утилиты man изучите утилиты *mknod, ls и grep*.
Научитесь в терминале выводить характеристики узла, зная его имя.

4.15.2 Структура винчестера и файловые системы

Прочитайте и усвойте учебный материал подразделов 1.2-1.6.
Закрепите изученный материал посредством практического использования утилит: *fdisk, mkfs, mknod, mount и umount*.

4.15.3 Стандартизация структуры ФС

Прочитайте и усвойте материал подраздела 4.7.
Запустите в окне терминала файловый менеджер Midnight Commander и с поиощью него изучите струтуры ФС всех трех уровней и директории */var*.
В окне терминала, усвойте работу команд (утилит) *pwd, cd, ls, cat*.
В домашней директории пользователя *upk*, усвойте работу утилит *mkdir u rmdir*.

4.15.4 Модули и драйверы ОС

Прочитайте и усвойте учебный материал подразделов 4.8-4.9.
В окне терминала исследуйте ветвь дерева директории */lib/modules*.
В окне терминала, усвойте работу утилит *insmod, rmmod, lsmod, modinfo u dmesg*.
Подробно изучите группировку и назначение функций системных вызовов ОС, представленных таблицей на рисунке 4.9.

5.15.5 Концепции работы с устройствами

Прочитайте и усвойте учебный материал подразделов 4.10-4.12.
Разберитесь в основных концепциях работы с устройтсвами (узлами) ОС.
С помощью руководства man и файла */etc/fstab* изучите способ задания конфигурации блочных устройств ОС.
Сравните структуру файла */etc/fstab* с выводом утилиты *mount*.
На примерах подраздела 4.12, освойте правильное использование утилит *mount u umount*. Особое внимание уделите атрибутам (опциям) монтирования.

4.15.6 FUSE и другие специальные ФС

Прочитайте и усвойте учебный материал подразделов 4.13-4.14.
На учебных примерах разберитесь с особенностями монтирования изученных ФС.
На практике освойте работу с утилитами: *df, du u lsof*.

Замечание

Примеры, приведенные в подразделе 4.3, являются чисто демострационными и не могут быть напрямую реализованы в ОС УПК АСУ.

4.15.7 Подключение рабочей области пользователя upk

На примере файла сценария */etc/upkasu/mount-upk.sh*, разберитесь и опишите в отчете алгоритм и функции монтирования рабочей области пользователя *upk*.

5 Тема 5. Управление пользователями ОС

Рассматривая базовые концепции ОС (подраздел 1.7), было отмечено, что *концепция пользователя* является второй по значимости после *концепции файла*.
С другой стороны, *концепция пользователя* интенсивно используется и в *концепции процесса*, что связывает их воедино, образуя три базовых концепции ОС.

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*, использующая ***имена*** и ***числовые идентификаторы***, распространяемая на все ПО ЭВМ;
- *система разграничений*, ***ограничивающая*** и ***специализирующая права*** использования ПО ЭВМ.

Кроме перечисленных выше, имеются специальные режимы использования ОС и ЭВМ, которые связаны с *настройкой, инсталляцией и промежуточными этапами загрузки ОС*.
Подобное многообразие взглядов порождает еще большее многообразие зависимостей, что влияет не только на работу отдельной ОС, но и распространяясь на социум, порождает новые технологические подходы к архитектуре современных ОС.

Чтобы успешно разобраться в указанных проблемах и освоить данную тему, учебный материал разбит на четыре части:

многопользовательский режим работы ОС, как альтернатива однопользовательскому, порождающий многобразие взглядов, уже указанных выше;
разграничение прав пользователей, как техническая основа функционирова-ния ОС;
login и система PAM, как реализация эффективного использования ПО ОС;
команды управления пользователями, как управляющая часть организации многопользовательского режима работы ОС.

5.1 Пользовательские режимы работы ОС

Первоначально, когда ОС еще не было, а использовалась пакетная обработка программ или программы супервизоры, понятие пользователя использовалось в чисто внешнем организационном плане: *как лицо*, передающее программу в службу ВЦ (вычислительного центра) на исполнение.
В 1974 году, Гари Килделл преподаватель информатики в аспирантуре военно-морского колледжа в Монтерее (Калифорния) закончил писать свою ОС для микрокомпьютера intel 8080, которую он назвал CP/M.
CP/M - Control Program for Microcomputers.
В 1980 году, фирма Microsoft приобрела у Seattle Computer лицензию на довольно

сырую и недоработанную операционную систему 86-DOS. Немного доработав, Билл Гейтс предложил использовать ее в качестве MS-DOS в первом персональном компьютере фирме IBM.
MS-DOS — доработанная 86-DOS фирмы Seattle Computer, способная запускать все программы ОС CP/M.
Характерная черта MS-DOS и CP/M — *однозадачный режим работы на изолированном компьютере*, что собственно говоря и не требовало ставить вопрос об использовании их многими пользователями. Эта черта в последующем отразилась и на ОС MS Windows:

- *автоматическое монтирование* всех файловых систем при старте ОС;
- *отсутствие концепции пользователя* в низкоуровневых структурах ФС.

История ОС UNIX начинается с середины 60-х годов, на фоне проекта операционной системы MULTICS, который разрабатывался в Bell Labs., подразделении гиганта AT&T.
Первое издание UNIX, появившееся в ноябре 1971 года, работало на PDP-11/20 без MMU и аппаратной защиты памяти. Стабильность ее работы и устойчивость к сбоям была не на высоте. *Мультипрограммности тоже не было*, но пути к файлам уже появились. Была документация к таким системным вызовам:

```
break, cemt, chdir, chmod, chown, close, creat, exec, exit, fork,
fstat, getuid,gtty, ilgins, intr, link, mkdir, mount,open, quit,
read, rele, seek, setuid, smdate, stat, stime, stty, tell, time,
umount, unlink,wait, write.
```

Из языков программирования поддерживались ассемблер, B, BASIC, FORTRAN. Языка C еще не было. Хотя явное упоминание о поддержке многопользовательского режима отсутствует, наличие команд *chown*, *getuid* и *setuid* говорит о том, что *в файловой системе концепция пользователя уже была заложена*.

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 изляции 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*.

Замечание

Практически всегда, под *именем пользователя* понимается *контекст*, соответствующий понятию *владелец*, который относятся применительно к файлам и процессам.

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
```

где

username	Имя пользователя, используемое для входа в систему. Содержит слово (ранее - до 8 букв). Заглавные буквы не допускаются.
password	Hash-код пароля. Сейчас ставится символ **x**, а hash-код пароля перенесен в файл ***/etc/shadow***.
UID	Число-идентификатор пользователя.
GID	Число-идентификатор основной группы, в которую входит пользователь.
GEOS	Любая информация.
homedir	Домашняя директория пользователя.
shell	Командный интерпретатор пользователя, который запускается при его входе в систему. Список возможных интерпретаторов находится в файле **/etc/shells**. Если имя пользователя не предназначено для интерактивной работы с ОС, то указывается **/bin/false** или ***/bin/nologin***

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
```

где

groupname	Имя группы с теми же ограничениями, что и имена пользователей.
password	Hash-код пароля (если пароль имеется). Сейчас ставится символ **x**, а hash-код пароля перенесен в файл ***/etc/gshadow***.
GID	Число-идентификатор группы.
userlist	Список пользователей, входящих в группу, разделенных запятыми. Первый пользователь в списке — администратор группы.

5.2 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.

На самом деле, имя пользователя имеет вспомогательное второстепенное значение. *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.

Замечание

Хотя часто, *при создании нового пользователя*, обычно создается и группа с таким же именем и GID=UID, - это не является обязательным требованием:

- при создании, новый пользователь может быть сразу включен в любую, уже существующую группу;
- любой пользователь, без ограничений, может быть включен в произвольное число групп.

Как правило, 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 достигпется командой:

```
su [-] [username]
```

при этом, ему придется ***набрать пароль*** того пользователя, под чьим именем он собирается работать:

- *Если присутствует первый аргумент* команды ***su***, то *произойдет смена домашней директории* и выполнятся скрипты входа типа ***~/.profile***.
- *Если второй аргумент не указан*, то подразумевается пользователь ***root***.

Выполнене команд от имени пользователя *root*, для обычного пользователя, выполняется командой:

```
sudo список_команд;
```

при этом, ему придется набрать *свой собственный пароль*.

Замечание

Если пользователю необходимо уточнить под чьими именами и идентификаторами он работает, можно воспользоваться командами ***whoami*** и ***id***.

Смена прав пользователя ОС связана с его действиями в системе.
Действия пользователя в системе определяется работой программ (процессов), которые пользователь запускает. Чтобы определить, с какими правами работает процесс, вводятся дополнительные понятия, связанные с *реальными и эффективными идетификаторами пользователя*.
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 десткриптора файла (см. подраздел 4.5, таблицы 4.9 и 4.10): если биты SUID и SGID - установлены, то эффективные ID берутся из дескриптора файла, а если нет, то устанавливаются в значения дейтсвительных (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(...)*.

Замечание

Установка новых значений эффектиных ID выполняется с помощью системных вы-зовов *setuid(...)* и *setgid(...)*.

На рисунке 5.1, представлен вывод эффектиных идентификаторов процесса shell,

запущенного пользователем upk в окне терминала.

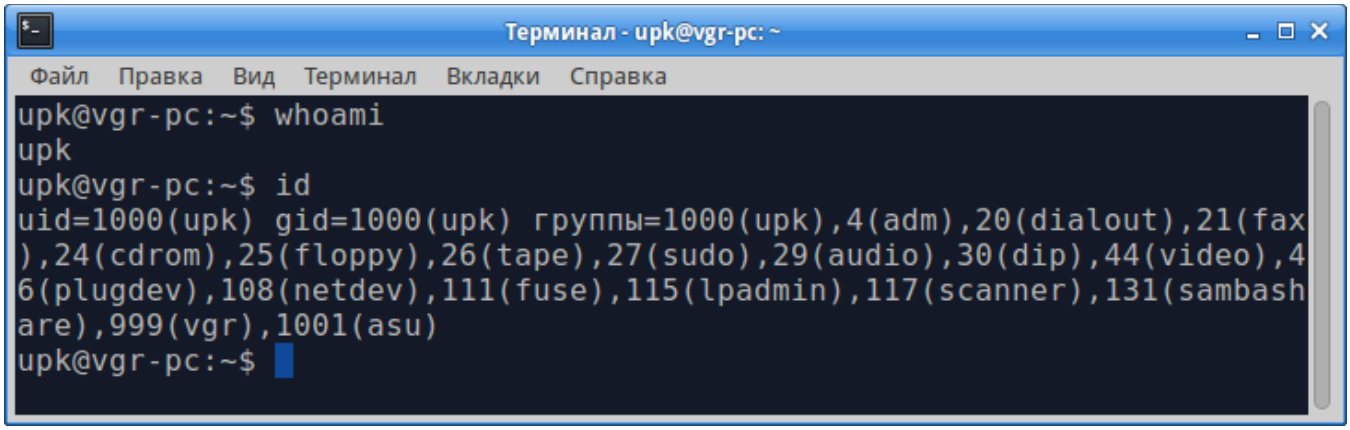

Рисунок 5.1 — Вывод эффективных ID пользователя upk

5.3 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.

После загрузки ОС и перехода ее в многопользовательский режим, запускается процесс «*Манеджер сеанса*», который контролирует и обеспечивает процедуры входа пользователя в систему:

- *в случае положительного завершения* процедры входа в систему, дополнительно контролируется содержимое директории /home/$USER и, случае необходимости, в нее добавляются нужные файлы или модифицируются старые, а также *устанавливаются эффективные идентификаторы* для процессов санса пользователя;
- *в случае негативного завершения* процедры входа в систему или *негативного контроля* содержимого директории /home/$USER, выполняются дополнительные процедуры, которые заканчиваются перезапуском «Менеджера сеансов».

Замечание

Среда исполнения, о которой говорилось ранее, представляет собой *одну систем-ную среду исполнения* и *пользовательские среды исполнения*, по одной на каждый вход в систему.

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 *процедура* ***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 процедуры 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.
В 1995 году, **OSF** (Open Software Foundation) — фонд открытого программного обеспечения - приступил к разработке *системы PAM* (Pluggable Authentication Modules) — *заменяемые модули идентификации*.

Замечание

К середине 90-х годов, проблема безопасности ОС стала столь критичной, что поставила под сомнение архитектурные принципы различных систем.
Система PAM разрабатывалась для UNIX-подобных систем.
MS Windows имеет свою оригинальную систему защиты.

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/x86_64-linux-gnu/security** или **/usr/lib/x86_64-linux-gnu/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/...**.

Замечание

Модульная система PAM не столько обеспечивает новый уровень защиты, сколько разделяет прикладную часть процессов от части, обеспечивающей защиту их функционирования, одновременно централизуя ПО защиты, обеспечивая мобильность его модификации и ускоряя внедрение новых технологий.

Технологическая концепция модулей PAM предполагает разделение их на четыре типа:

auth	Выполняют аутентификацию, то есть подтверждают, что пользователь является именно тем, кем он представился в системе.
account	Разрешают или запрещают конкретному пользователю вход в систему. Это решение может зависеть от даты, времени суток, системных ресурсов и т.д.
session	Осуществляют действия, которые должны быть выполнены до или после входа пользователя: занесение информации в журнальные файлы, монтирование устройств и т.д.
passwd	Изменяют пароль пользователя.

Практическая реализация этой концепции предполагает централизованное исполь-зование файлов конфигурации.
Сейчас, файлы конфигурации Linux-PAM находятся в директории */etc/pam.d*, в которой находятся файлы, как правило совпадающие с именами файлов приложений. Например, для программы *sudo*, файл конфигурации: */etc/pam.d/sudo*.
Каждый файл конфигурации состоит из отдельных строк, содержащих поля:

```
тип_модуля    управляющий_флаг имя_модуля    аргументы
```

тип_модуля	Один из четырех типов: auth, account, session, passwd.
управляющий_флаг	Флаг, контролирующий поведение PAM в случае успешного или безуспешного результата работы модуля: • *required* - успешное завершение работы модуля необходимо для успеха всего запроса. Об ошибочном завершении не будет сообщено до окончания работы всех модулей. • *requisite* — успешное завершение работы модуля необходимо для успеха всего запроса. Ошибочное завершение приводит к немедленному возврату управления приложению. • *sufficient* — в случае успешного завершения, управление немедленно передается приложению. Ошибочное завершение модуля не учитывается. • *optional* — результат работы этого модуля не учитывается.
имя_модуля	Имя файла, которое должно быть указано с полным путем к нему.
аргументы	Командная строка, передаваемая модулю.

Таким образом, система PAM позволяет разрабатывать систему безопасности без переделки самих приложений.

Замечание

Чтобы узнать, *какие библиотеки* PAM использует приложение, *лучше* воспользоваться командой **ldd**. Например, для приложения ***su***, имеем:

```
$  ldd /bin/su
     linux-gate.so.1 =>  (0xb76e8000)
     libpam.so.0 => /lib/i386-linux-gnu/libpam.so.0 (0xb76c8000)
     libpam_misc.so.0 => /lib/i386-linux-gnu/libpam_misc.so.0 (0xb76c4000)
     libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7519000)
     libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb7514000)
     /lib/ld-linux.so.2 (0xb76e9000)
$
```

5.4 Команды управления пользователями

Теоретическая концепция пользователя поддерживается соответствующей информационной и управляющей инфраструктурой ОС.
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*.

Замечание

Прежде чем выполнять любую из этих команд, следует воспользоваться руководством ***man***, а после — предварительно запустить команды с ключем --***help***.

Общий синтаксис команды *создания нового пользователя* имеет вид:

```
    useradd [ -A { DEFAULT | method [ ,... ] } ]
        [ -c comment ]
        [ -d home_dir ]
        [ -e expire_date ]
        [ -f inactive_time ]
        [ -g initial_group ]
        [ -G group [ ,... ] ]
        [ -m [ -k skeleton_dir ] | -M ]
        [ -s shell ]
        [ -u uid [ -o ]] имя_пользователя
        [ -r ]
        [ -n ]
```

Команда требует одного обязательного аргумента — *имя_пользователя*. Многие другие аргументы команды интуитивно понятны и не требуют пояснения.

Замечание

Наличие множества параметров команды создания пользователя требует хорошего навыка и знания как пользователь работает. Каждый администратор ОС вырабатывает свои основные правила работы с пользователями, поэтому использует только часть ее возможностей. Следовательно, после ввода, команда может задать вопросы в интерактивном режиме. Естественно имеются графические утилиты, которые создают пользователя по умолчанию и позволяют редактирование их атрибутов в более удобном виде.

Удаление пользователя выполняется командой:

```
userdell имя_пользователя;
```

Если пользователь с таким именем существует и, при этом, не находится в системе, то *userdell*:

- *удаляет его домашний каталог* со всеми подкаталогами;
- *удаляет все записи* об этом пользователе из файлов */etc/passwd, /etc/shadow, /etc/group*;
- *возможно, оставляет* временные забытые файлы в директории */tmp*.

Изменение параметров пользователя выполняется командой *usermod*, в которой большинство аргументов совпадают с аргументами команды *useradd*, но ориентированы на изменение соответствующих параметров.

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

- *администратор ОС* может изменить пароль любого пользователя;
- *работа утилиты*, практически всегда происходит в интерактивном режиме; например, новый пароль вводится дважды;
- *обычный пользователь* может сменить только свой пароль, предварительно набрав старый.

Общий синтаксис команды:

```
passwd [параметры] [LOGIN]
```

Замечание

Обязательно следует изучить параметры команды с помощью руководства ***man passwd*** и запуска ее с ключем —***help***.

Утилита passwd имеет множество опций, поэтому рассмотрим наиболее типичные варианты ее применения.

```
passwd [ -f | -s ] [ имя ]
```

-f Позволяет изменить поле GEOS в файле ***/etc/passwd***.

-s Позволяет изменить интерпретатор ***shell***, вызываемый при входе пользователя в систему.

```
passwd [ -g ] [ -r | -R ] группа
```

-g Переключает passwd в режим работы с паролями групп.

-r Удаляет групповой пароль.

-R Закрывает доступ к группе для всех пользователей.

```
passwd [ -x max ] [ -n min ] [ -w warn ] [ -i inact ] имя
```

-x max Максимальное число дней, в течение которых пароль действителен. 9999 — пароль действителен всегда.

-n min В течение скольких дней пользователь не может изменять свой пароль. 0 — может всегда.

-w warn За сколько дней до истечения срока max пользователю начнут выдаваться предупреждения о необходимости смены пароля.

-i inact Число дней, свыше max, когда пользователь может сменить пароль, иначе он будет заблокирован до вмешательства администратора.

```
passwd { -l | -u | -d | -S } имя
```

-l Временно запретить доступ пользователя в систему.

-u Восстановить доступ пользователя в систему.

-d Удаление пароля пользователя с разрешением входа в систему (без пароля).

-S Получить информацию о пароле пользователя. Например,

```
$  passwd -S
vgr P 12/21/2012 0 99999 7 -1
$
```

5.5 Лабораторная работа по теме №5

Цель лабораторной работы №5 — практическое закрепление учебного материала по теме «Управление пользователями ОС».
Метод достижения указанной цели — закрепление учебного материала, изложенного в первом разделе пособия посредством утилит ОС, а также выполнение заданий, приведенных в данном разделе.
Чтобы успешно выполнить данную работу, студенту следует:

- *запустить с flashUSB* ОС УПК АСУ, подключить личный архив и переключиться в сеанс пользователя ***upk***;
- *запустить на чтение* данное пособие и на редактирование личный отчет;
- *открыть одно или несколько окон терминалов*, причем хотя бы в одном окне терминала открыть Midnight Commander, для удобства работы с файловой системой ОС;
- *приступить к выполнению работы*, последовательно пользуясь рекомендациями представленных ниже подразделов.

Замечание

Многие команды ОС студенту еще не известны, поэтому следует:

- для вывода на консоль руководства по интересуемой команде, использовать: ***man имя_команды***;
- для выяснения сущесвования команды, ее доступности и местоположения, использовать: ***command -v имя_команды***;
- для уточнения правил запуска конкретной команды, можно попробовать один из вариантов: ***команда --help*** или ***команда -h*** или ***команда -?***.

В процессе выполнения лабораторной работы студент заполняет личный отчет по каждому изученному вопросу!.

5.5.1 Инфраструктура управления пользователями

Прочитайте и усвойте учебный материал подраздела 5.1.
Исследуйте содержимое директорий: */etc/passwd, /etc/shadow, /etc/group, /etc/gshadow*.
Усвойте структуру и назначение этих файлов.

5.5.2 Реалные и эффективные права пользователя

Прочитайте и усвойте учебный материал подраздела 5.2.
С помощью руководства man изучите утилиты whoami, id, chown и chmod.
С помощью учебного материала подраздела 4.5, а также таблиц 4.9 и 4.10, изучите структуру поля дескриптора файлов i_mode.
Усвойте назначение битов SUID и SGID.
В директории *~/src* создайте текстовый файл **test** и включите в него команды *id* и

whoami.
Сделайте файл *~/src/test* исполняемым и, запуская его, исследуйте эффективные идентификаторы запускаемого процесса.
Находясь в директории ~/src, уснановите значения битов SUID и SGID командой:

```
sudo chmod 3777 ./test
```

Запустите команды:

```
./test
sudo ./test
```

Сравните результаты.

5.5.3 Инфраструктура PAM

Прочитайте и усвойте учебный материал подраздела 5.3.
Исследуйте содержимое директории */etc/skel* и сравните с содержимым рабочей директории пользователя **upk**.
Изучите содержимое директории */lib/x86_64-linux-gnu/security*.
Изучите содержимое директории */etc/pam.d* и файла */etc/pam.conf*.
Изучите утилиту ***ldd*** и исследуйте с помощью нее ряд утилит, которые вы считаете, участвуют в контроле прав доступа пользователей.

5.5.4 Команды управления пользователями

Прочитайте и усвойте учебный материал подраздела 5.4.
Изучите утилиты **useradd**, **userdel** и **usermod**.
Из главного меню рабочего стола откройте окно «Все настройки», выберите и запустите ПО, озаглавленное «Пользователи и группы», как показано на рисунке 5.2.
Добавьте нового пользователя, например с именем *mmm*.
Исследуйте содержимое директорий: */etc/passwd, /etc/shadow, /etc/group, /etc/gshadow*.
Запустите Midnight Commander в окне терминала и перейдите в директорию */home/mmm*.
Сравните содержимое директории /home/mmm с содержимым диреткоии */etc/skel*.
Закройте все окна пользователя **upk** и выйдите из его сеанса.
Войдите в сеанс пользователя **mmm**, запустите Midnight Commander в окне терминала и исследуте содержимое директории */home/upk*.
Выйдите из сеанса пользователя mmm и зайдите в сеанс пользователя **upk**.
Запустите главное окно работы с пользователями и удалите пользователя **mmm**.
Исследуйте изменения структуры файлов и директорий.

Замечание

Учитывая, что настоящий дистрибутив не содержит графической утилиты управления пользователями, выполните данный пункт лабораторной работы с помощью утилит:
useradd, **userdel** и **usermod**.

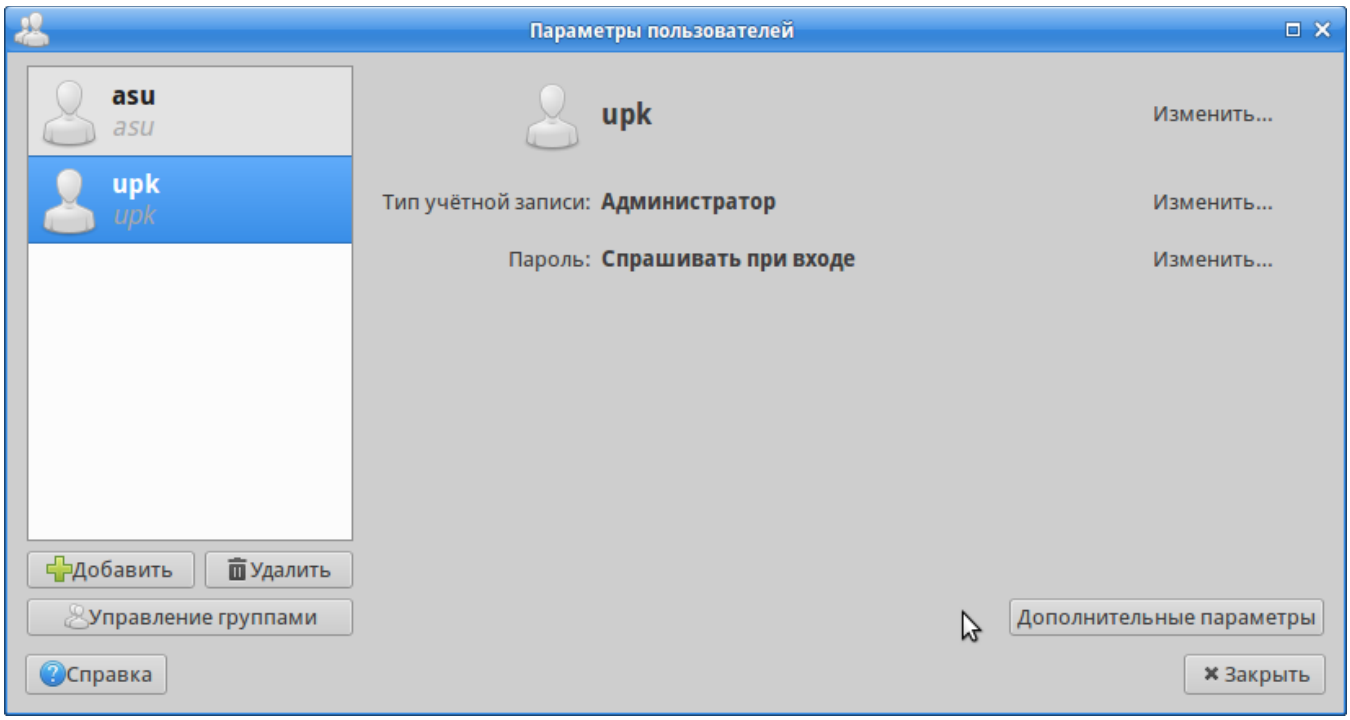

Рисунок 5.2 — Главное окно работы с пользователями ОС

Завершив выполнение всех заданий и оформление отчета:

- провести архивирование и сохранение рабочей области **upk**;
- выключить компьютер и завершить выполнение лабораторной работы №5.

6 Тема 6. Теоретическая часть

Учебный материал, изложенный в этой части пособия, последовательно раскрывает понятие процесса, которое уже было дано раньше в предыдущих темах.
Прежде всего, следует вспомнить три базовых концепции ОС, изученных нами в первой теме (подраздел 1.7), и повторно обратиться к рисунку 1.10, который повторно представлен на рисунком 6.1 и наглядно отражает взаимодействие этих концепций.

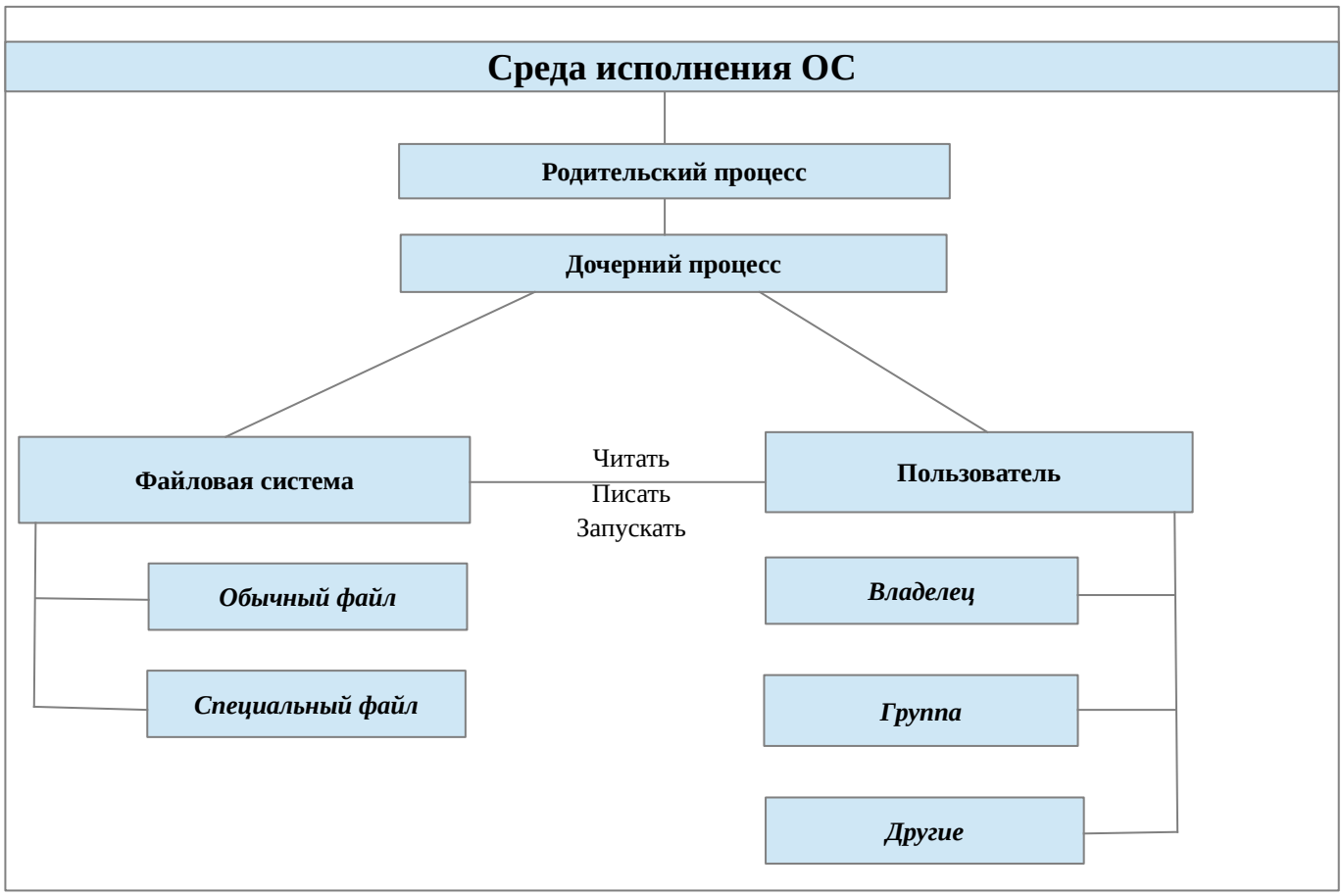

Рисунок 6.1 - Базовые концепции пользовательского режима ОС

В предыдущих двух темах (см. разделы 4 и 5), мы подробно изучили концепции файла и пользователя, а также закрепили этот материал на практике выполнения соотвествующих лабораторных работ. Здесь мы завершим начатое ранее, разобрав все подробности концепции процесса применительно к функционированию ОС.

6.1 Подсистема управления процессами

Вспомним определение процесса, данное ранее в подразделе 1.7.
Процесс — это элементарный управляемый объект ОС, имеющий целочисленный идентификатор *PID - Process Identification*, обеспечивающий функциональное преобразование файлов (данных) с правами, которые определяются объектами *пользо-*

ватель.

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.

Чтобы определить место концепции процесса в архитектуре ПО ОС, обратимся к рисунку, который также был рассмотрен в первой теме (подраздел 1.5, рисунок 1.5), и повторно представлен на рисунке 6.2.

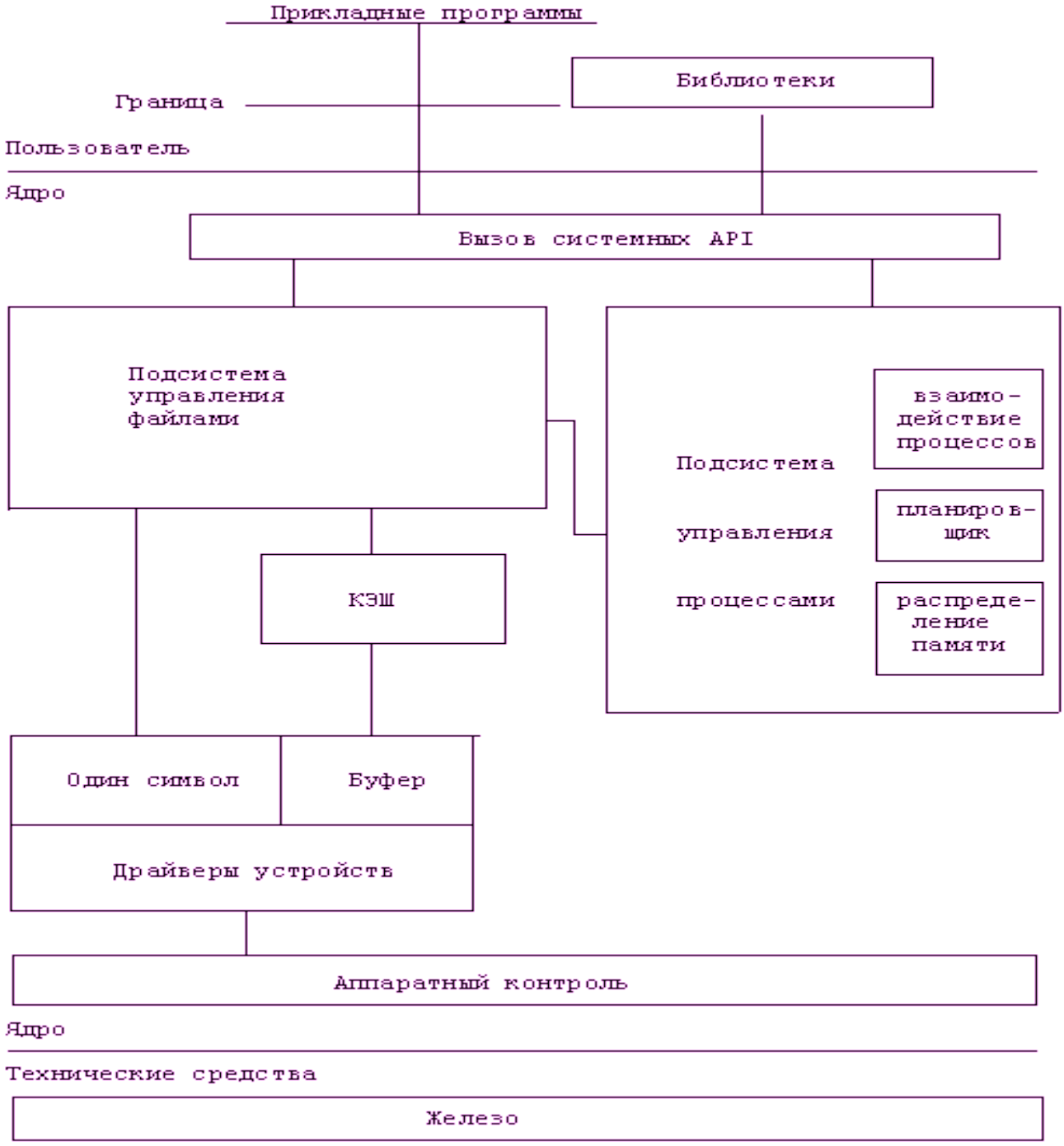

Рисунок 6.2 - Классическая архитектура ПО ОС UNIX

Хорошо видно, что:

- *все процессы* находятся в среде пользователя (*функционируют в пользовательском режиме*);
- *ядро ОС* имеет *подсистему управления процессами*, которая функционирует в защищенном (привелигированном) режиме процессора.

С другой стороны, в самой подсистеме управления процессами выделяются три подсистемы:

- взаимодействие процессов;
- планировщик;
- распределение памяти.

Таким образом, мы видим, что управление процессами — очень сложная часть деятельности ОС, включающая *два противоположно направленных воздействия*:

- ***процесс воздействует*** на подсистему управления процессами напрямую, через API ядра ОС посредством системных вызовов, или косвенно, через подсистему управления файлами;
- ***подсистема управления процессами воздействует*** на процессы посредством трех выделенных механизмов: взаимодействия процессов, планировщика и средств распределения памяти.

Все указанные аспекты взаимодействия будут рассмотрены в данной теме, а начнем изучение с пользовательского режима работы ОС, в котором создается и начинает функционирование «*Главный родительский процесс*».

6.2 Главный родительский процесс init

Понятие процесс является третьей базовой концепцией ОС, которая опирается на концепции *файла* и *пользователя*:

- *процесс создается клонированием* из родительского процесса, получая от ядра ОС целочисленный номер, уникально идетифицирующий его для целей управления им;
- *процесс наследует* все права пользователя от родителя, включая все доступные на момент создания ресурсы родителя;
- *процесс управляется родителем*, который ожидает завершение его функционирования или посылает сигналы, частично интерпретируемые самим процессом, частично - ядром ОС;
- *процессу разрешается модифицировать себя*, открывая новые ресурсы и закрывая старые, а также загружая, в свое пространство из файлов, *новые алгоритмы функционирования* и *новые права пользователя*;
- *процессу разрешается создавать дочерние процессы*, тем самым, выполнять функции родительского процесса.

Таким образом, указанные свойства процесса закладывают теоретические основы *мультипрограммного режима работы* ОС, который является концептуальной основой практически всех современных систем.

Мультипрограммный режим работы ОС предполагает, что ядро ОС организует *параллельное выполнение множества процессов на одном компьютере*. При этом, процессы могут никак не взаимодействовать друг с другом.
Таким образом, **процесс** — это элементарная защищенная единица выполняемого в режиме пользователя программного кода, которым управляет ядро ОС, предоставляя ему системные ресурсы компьютера в виде процессорного времени и обмена данными с внешними устройствами.

Другой аспект концепции процесса, - *понятие мультизадачного режима работы* ОС.
Мультизадачный режим работы ОС предполагает, что в пространстве пользователя организуется параллельное выполнение вычислений, посредством одного или нескольких процессов, с использованием специальных механизмов взаимодействия: *синхросигналы*, *семафоры* и *передачу данных*.
Для реализации такого режима были введены понятия *легковесных процессов* (thin), которые теперь называют: *потоки выполнения*, *нити* или *треды* (threads).

Таким образом, хотя понятия мультипрограммного и мультизадачного режимов работы ОС - во многом похожи, между ними имеется существенное различие:

- *мультипрограммый режим* - средство распараллеливания работы процессов;
- *мультизадачный режим* — средство объединения процессов по их прикладному назначению.

Поскольку, все процессы достаточно сильно взаимодействуют через механизм наследования, то для разделения процессов на максимально независимые группы по прикладному назначению используется специальный прием:

- *главный процесс задачи* создает *клон* (дочерний процесс) и завершает свою работу, не дожидаясь завершения работы дочернего процесса;
- *клон продолжает функционировать*, становясь дочерним процессом главного родительского процесса с номером PID=1;
- *клон, при необходимости*, порождает свои дочерние процессы, становясь главным процессом прикладного назначения (задачи).

Замечание

Когда родительский процесс завершил свою работу не дождавшись корректного завершения своих дочерних процессов, возникает такое явление как *процессы зомби*.

Зомби — процесс, прекративший свое целевое функционирование, но в силу своего некорректного взаимодействия с породившим его и завершившимся родительским процессом:

- *перешел под управление* главного родительского процесса с PID=1;
- *находится под управлением ядра ОС*, поскольку его PID не удален из системы.

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.

Особая важность процесса init делает его объектом постоянных исследований и усовершенствований. Более подробно этот аспект мы изучим в подразделе 6.6 - «**Четыре подхода к управлению процессами**». Здесь же, мы отметим разные подходы по его созданию ядром 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, взаимодейтсвуя 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* создаетя среда для файла */sbin/init*;
- *переключение ядра в пассивный режим* осуществляется по классической схеме;
- *работа ядра в пассивном режиме* осуществляется по классической схеме.

Замечание

Из сказанного следует, что первый процесс не обязательно должен иметь имя **init,** но по традиции это продолжают делать, следуя классике изложения, а другие варианты рассматривают как альтернативные.

6.3 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*, показанная на рисунке 6.2.
Планировщик - часть ПО ядра ОС, который организует *некоторую очередь* процессов и переключает их в *различные состояния*, обеспечивая мультипрограммный режим их функционирования.

Замечание

Переключение процессов в различные состояния осуществляется не только с помощью программных средств ОС. Широко используются аппаратные средства процессора, которые называются переключением задач, *обеспечивая мультизадачный режим его работы*, что не следует путать с *мультизадачным режимом работы ОС*.

Различные реализации планировщиков отличаются прежде всего *количеством состояний процесса*, которые он учитывает. На рисунке 6.3 показана схема простейшего планировщика, учитывающего *только два состояния кажного процесса*. Она очень хорошо соответствует системе управления с обратной связью.

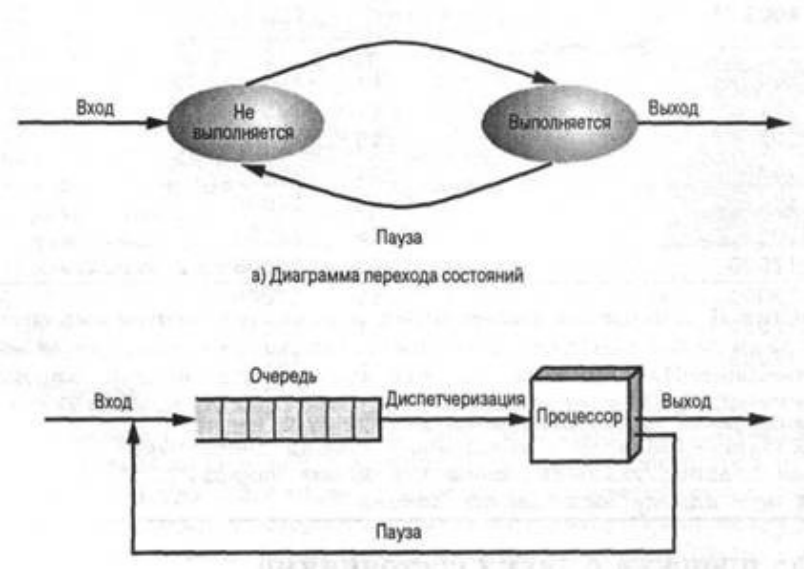

Рисунок 6.3 — Модель планирования с двумя состояниями

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

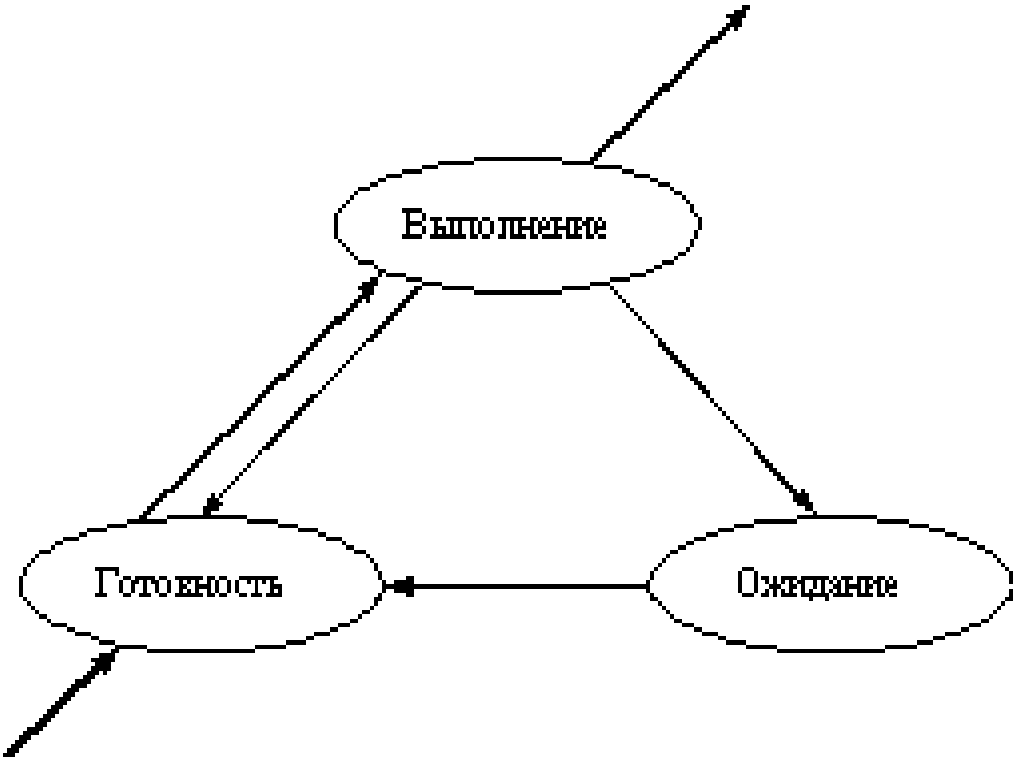

Рисунок 6.4 — Классическая модель планирования с тремя состояниями

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.

Обе схемы, показанные на рисунках 6.3 и 6.4, предполагают, что процесс существует в системе и его выполнение может быть многократно прервано и продолжено.
Чтобы обеспечить прерывание и продолжение процесса, реализуются структуры данных, которые соотносятся с понятиями *контекста* и *дескриптора процесса*, а также — с *очередью процессов*.
Контекст процесса — данные о состояние операционной среды, отображаемые состоянием регистров и программного счетчика процессора и режимом его работы, указателями на открытые файлы ОС, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и другими подобными сведениями.

Дескриптор процесса — более оперативная и конкретная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента процесса и другая информация.

Очередь процессов - дескрипторы отдельных процессов, *объединенные в списки*, таким образом, что каждый дескриптор содержит по крайней мере один указатель на другой дескриптор, соседствующий с ним в очереди.

Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

Таким образом, обеспечить планирование процесса - это значит:

- *создать информационные структуры*, описывающие данный процесс: его дескриптор и контекст;
- *включить дескриптор нового процесса* в очередь готовых процессов;
- *загрузить кодовый сегмент процесса* в оперативную память или в область свопинга.

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 соотвествует полному «жизненному циклу процесса».

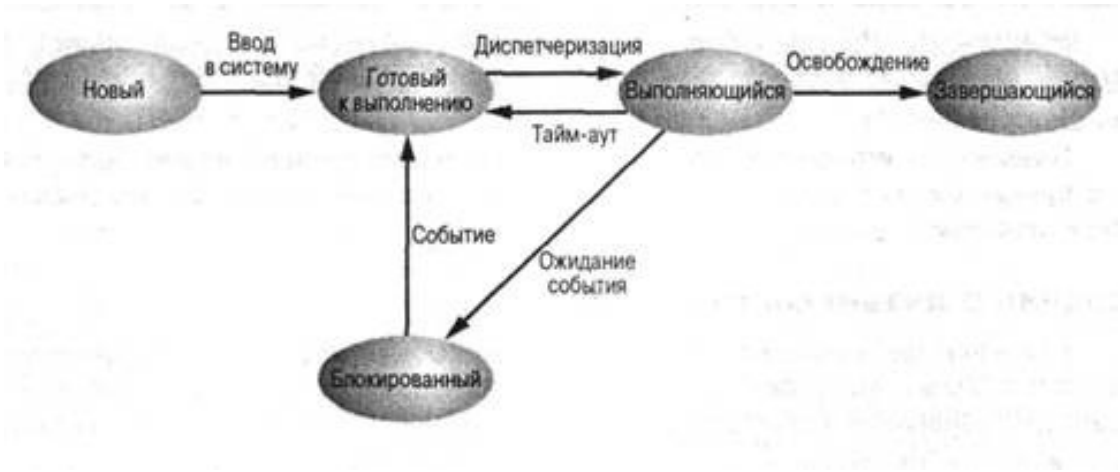

Рисунок 6.5 — Модель планирования с пятью состояниями

Замечание

Добавление состояний создания и завершения процесса безусловно позволяет строить более сложные и эффективные алгоритмы планирования, но их значение является второстепенным, по сравнению с тремя классическими состояниями.

Целенаправленное обеспечение «жизненного цикла процессов» называется *стратегией диспетчеризации (стратегии планирования)*.

Все стратегии планирования процессов предназначены для достижения *различных целей* и эффективны для *разных классов задач*.

Различные ОС ориентированы на *различные классы задач* и реализуют *различные среды исполнения процессов*.

В общем случае, можно выделить три среды, которые формируют ОС:

- Системы пакетной обработки данных;
- Интерактивные системы;
- Системы реального времени.

6.4 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*.

6.5 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.

Существует множество различных алгоритмов планирования процессов, по разному решающих вышеперечисленные выше задачи и преследующих различные цели, что обеспечивает различное качество мультипрограммирования.
Среди этого множества алгоритмов 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.
Граф состояний процесса, при таком подходе, соответствует классической модели состояний, показанной ранее на рисунке 6.4.
Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в *системах разделения времени*.

Замечание

Традиционно, ОС UNIX и Linux относятся к *системам разделения времени*.
Разработчики ориентировали эти ОС на создание *среды пакетной обработки данных и среды интерактивных систем*:

- максимальная пропускная способность решения задач в единицу времени;
- минимизация времени на ожидание обслуживания и обработку задачи;
- поддержка постоянной занятости процессора;
- быстрая реакция на запросы;
- выполнение пожеланий пользователя.

Кванты времени, выделяемые процессам, могут быть:

- *одинаковыми* для всех процессов или *различными*;
- *фиксированной величины* для одного процесса или *изменяться* в разные пери-

оды жизни процесса.

Процессы, которые не полностью использовали выделенный им квант, например, из-за ухода на выполнение операций ввода-вывода, могут получить или не получить *компенсацию в виде привилегий* при последующем обслуживании.

По разному может быть организована очередь готовых процессов:

- *циклически*, по правилу "первый пришел - первый обслужился" (FIFO);
- *по правилу стека* - "последний пришел - первый обслужился" (LIFO).

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, что показано вариантом а) на рисунке 6.6;
- *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, что показано вариантом б) на рисунке 6.6.

Замечание

Во многих ОС алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

Примениельно к алгоритмам, существует *два типа процедур планирования*:

- *вытесняющая* многозадачность;
- *невытесняющая* многозадачность.

Preemptive multitasking - *вытесняющая многозадачность* - способ, при котором решение о переключении процессора, с одного процесса на другой, принимается планировщиком операционной системы, а не самой активной задачей.

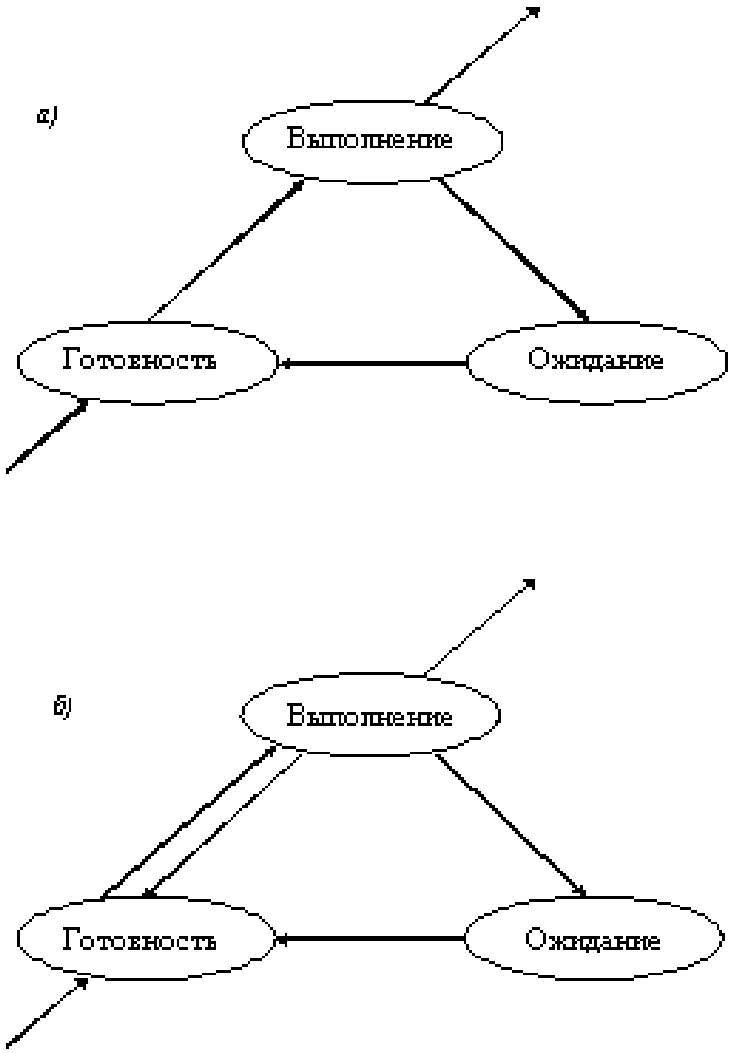

Рисунок 6.6 — Переходы состояний для относительного и абсолютного приоритетов

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

Реализация алгоритмов разделения времени, как и других алгоритмов планирования, сталкивается с существенными проблемами, что делает их достаточно сложными:

- *проблема синхронизации* - задача борьбы процессов за один ресурс, рассматриваемая в последующих подразделах, на примере *разделяемой памяти*;

- *проблема тупиков* - бесконечное ожидание ресурса, который занят и не освобождается другим процессом; *этот вопрос выходит за рамки первой части данной дисциплины*.

6.6 Четыре подхода к управлению процессами

Рассмотрев работу ядра ОС на уровне стратегии и алгоритмов планирования мультипрограммного режима функционирования, вернемся к «*Главному родительскому процессу*», который организует управление всеми процессами, функционирующими в пользовательском режиме.

По традиции, этот процесс назван ***init***, что никоим образом не ограничивает его возможности, а является следствием решения проблемы переключения режимов работы ядра ОС: *из активного* режима *в пассивный*.
Более того, создаваясь первым, этот процесс является:

- *родительским процессом* для всех остальных процессов;
- *передает дочерним процессам* свою среду исполнения и непосредственно управляет процессами своего ближайшего окружения;
- *отслеживает завершение всех процессов* в режиме пользователя и становится прямым родителем процессов, родители которых завершили свою работу (*процессы зомби*);
- *завершает работу последним*, одновременно завершая работу ядра ОС.

Таким образом, с процесса ***init*** начинается *организация* и *управление* системным и прикладным программным обеспечением компьютера.

Принципиально, в качестве ***init*** может быть запущена *любая программа*, содержащая *любой алгоритм*.
В частности ***init***, с помощью системного вызова *exec(...)*, может вместо себя загрузить любую другую программу, которая будет выполнять роль «*Главного родительского процесса*».
В общем случае, можно организовать цепочку таких программ, *порождая различные подходы* к управлению процессами.

Практика использования ОС потребовала *унификации подходов* к общей организации вычислительного процесса компьютера и, в первую очередь, *разделение работы системного и прикладного ПО*.
Мы рассмотрим наиболее известные четыре подхода: ***монопольный***, ***System V***, ***upstart*** и ***systemd***.

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 управ-

ления 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.

Например, ОС Linux, используя универсальный загрузчик *GRUB*, указывает в файле *grub.cfg* местоположение ядра ОС (*vmlinuz*) и файл *initrd* — сжатой временной файловой системы ОС.
GRUB обеспечивает:
- загрузку и распаковку ядра ***vmlinuz***;
- загрузку файла ***initrd***;
- передачу ядру параметров, указанных в ***grub.cfg***, и координат ***initrd***;
- передачу управления ядру ОС перед завершением работы.

Ядро ОС:
- инициирует внутренние параметры;
- распаковывает в память ЭВМ временную ФС ***initrd***;
- находит в корне временной ФС файл ***init*** или ***linuxrc***;
- создает первый процесс ***init*** или ***shell***, если ***init*** — скрипт.

Такой подход обеспечивает широкие возможности по созданию собственных дистрибутивов ОС, ориентированных на специальные приложения.

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.
UNIX System V — одна из версий ОС UNIX, разработанная компанией ***AT&T*** и выпущенная **в 1983 году**. Всего было выпущено четыре версии. Версия UNIX System V Release 4 (***SVR4***) была наиболее удачной. Разработчики SVR4 выпустили стандарт — *System V Interface Definition* (**SVID**), описывающий работу этой ОС.
Многие разработчики UNIX-подобных систем стали ориентироваться на этот стандарт. В частности, переняли от нее сценарии инициализации системы «*SysV init scripts*», отвечающие за запуск и выключение ОС. Такие сценарии традиционно находятся в директории **/etc/init.d/**.

Замечание

К концу 90-х годов, значение SVID снизилось и были выпущены, независимые от производителя ОС, стандарты POSIX и Single UNIX Specification (SUS).
ОС Linux не сертифицирует свои дистрибутивы и использует, основанный на SUS, стандарт *Linux Standard Base* (*L*SB).

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.

Администратор ОС может установить нужный уровень, используя утилиту:

```
telinit новый_уровень_ОС;
```

Таблица 6.1 - Уровни работы ОС

Уровень	***Назначение***
0	Выключение системы.
1	Однопользовательский режим (для администрирования системы).
2 - 4	Нормальная работа (настраивается администратором).
5	Нормальная работа (запускается X Window System).
6	Перезагрузка ОС.

Процесс ***init*** «знает» об уровнях и, после запуска, читает файл */etc/inittab*, структура которого имеет вид отдельных строк (пустые строки и строки, начинающиеся с символа # - игнорируются):

```
id:runlevels:action:process
```

где

id	0-4 символов, уникально идентифицирующих строку.
runlevels	Список уровней *из таблицы 6.1* (без разделителей), на которые действует строка.
action	Одно из спецификаций действия на **process**, перечисленные *в таблице 6.2*, и которые конкретный **init** должен понимать.
process	Выполняемая команда.

Таблица 6.2 - Спецификации действий

Действие	***Описание***
respawn	Процесс будет запущен снова, если он завершился.
wait	Процесс будет запущен один раз и **init** будет ждать его завершения.
once	Процесс будет запущен один раз.
boot	Процесс будет запущен при загрузке ОС. Значение **runlevels** — игнорируется.
off	Ничего не делать.
initdefault	Определяет ***уровень по умолчанию***, но котором запускается ОС.
sysinit	Процесс будет запущен во время ОС и перед спецификацией **boot**.
ctrlaltdel	Процесс запустится, когда нажаты клавиши **Ctrl-Alt-Del**.

Для примера, рассмотрим абстрактный файл */etc/inittab* (ОС *Knoppix* клон *Debian*):

```
id:5:initdefault:

# Boot-time system configuration/initialization script.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:respawn:/bin/bash -login >/dev/tty1 2>&1 </dev/tty1

l0:0:wait:/etc/init.d/knoppix-halt
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/knoppix-reboot

# Run X Window session from CDROM in runlevel 5
w5:5:wait:/bin/sleep 2
x5:5:wait:/etc/init.d/xsession start
```

Замечание

Для любой ОС, использующей систему инициализации *System V*, следует отдельно изучить возможности процесса ***init***, структуру файла */etc/inittab* и директории, в которых находятся скрипты инициализации уровней.

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*.

Широкое использование в UNIX-Linux системах графических режимов работы *ОС*, сильно обострило сложившуюся ситуацию.
Стремление реализовать технологию *Plug and Play* и обеспечение надежной работы графической системы *X Window System*, широко использующих *технологию событий*, была предложена ***системная шина D-Bus***.

D-Bus - *высокоуровневая система межпроцессного взаимодействия*, обеспечивающая *универсальный сервис* взаимодействия прикладных процессов в системе.

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/**.

Во время загрузки ОС, новая /sbin/init:

- *запускает сразу все приложения*, для которых имеются файлы конфигурации;
- *если приложение не может запуститься* по причине отсутствия некоторых ресурсов или еще не запущенных других приложений, оно переводится в режим ожидания предоставления таких ресурсов с помощью сигналов;
- *сигналы старта и останова приложений* принимаются ***init*** через шину ***D-Bus*** и используются для принятия действий, описанных в файлах конфигурации.

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.

Для отображения разновидности работ, в файлах конфигурации используются специальные ключевые слова:

- **exec команда** — для запуска отдельного приложения;
- **script …... end script** — операторные скобки скрипта shell;
- **respawn** — ключевое слово для обозначения сервиса;
- **task** — ключевое слово для обозначения задачи;
- **manual** — ключевое слово для обозначения работы, которая будет запускаться и останавливаться в ручную.

Для управления работами используется утилита **initctl**, например:

initctl list — позволяет просмотреть все запущенные сервисы;
initctl version — позволяет узнать версию upstart;
initctl start имя_сервиса — стартует сервис;
initctl stop имя_сервиса — останавливает сервис.

Таким образом, система инициализации **upstart** снимает *проблему следования строгому порядку запуска всех процессов*, ограничивая ее проблемами приложений, ограниченных одним сервисом.
Приложения, ограниченные одним сервисом, объединяются в понятие ***cgroups***, работу которых и отслеживает upstart.

Успехи upstart, естественным образом, приводят к мысли об универсальном подходе к управлению процессами и устройствами ЭВМ.

С апреля 2010 года стартовал авторский проект (Леннарт Поттеринг), названный ***systemd***.
Systemd - *демон инициализации*, призванный унифицировать управление устройствами и процессами, заменив существующие программные средства, включая ***init***.

Sytemd 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.

Многие дистрибутивы Linux — ***Fedora***, ***Mageia***, ***Mandriva***, ***Rosa***, ***OpenSUSE*** - уже используют systemd.
Википедия дает следующее представление о компонентах systemd (см. рисунок 6.7):

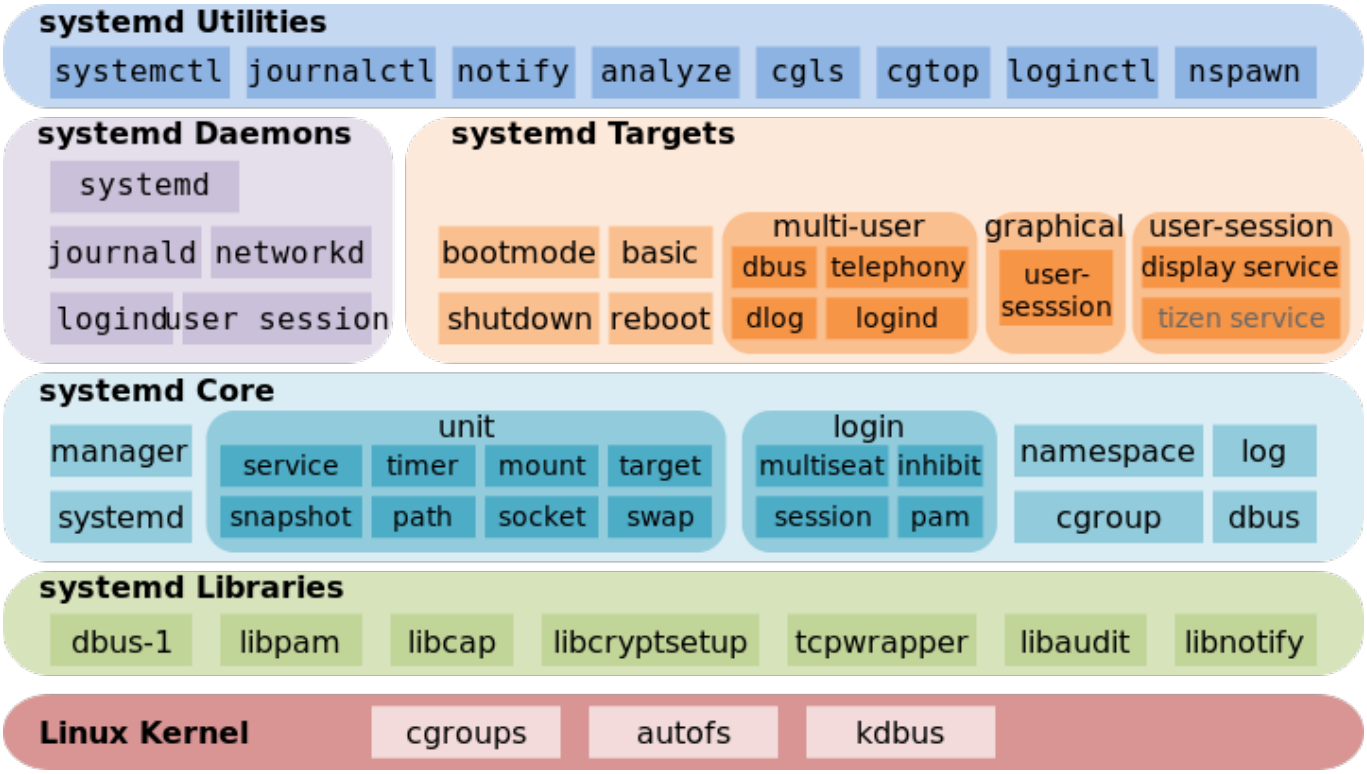

Рисунок 6.7 - Компоненты systemd (Википедия).

Замечание

ОС Ubuntu и ее клоны используют systemd лишь — *опционально* - для управления устройствами, *возлагая управление процессами на upstart*.
ОС УПК АСУ использует systemd, включая предыдущее замечание.

6.7 Стандарты POSIX и сигналы

Любое управление связано с взаимодействием.
Базовые способы управления процессами:

- *порождение процесса* с помощью системного вызова *fork(...)* и передача ему алгоритмического содержания в виде кода и всех открытых ресурсов родительского процесса;
- *изменение алгоритмического содержимого процесса* и дополнительных данных посредством системного вызова *exec(...)*.

Развитие способов управления процессами начинается с *сигналов*.

С прикладной точки зрения, *каждый процесс* — это прикладная программа, выполняющая расчетную часть некоторой задачи или обеспечивающая решение этой задачи.
Для ОС, ***процесс*** - это функциональный элемент, *требующий ресурсов* процессора, оперативной памяти, файловой системы или некоторых средств коммуникации.
Необходимые ресурсы вычислительной системы (комплекса), определяются алгоритмом решения задачи и потребностями самой задачи в этих ресурсах, что приводит к *конкуренции между процессами*.
В конечном итоге, конкуренция между процессами приводит к захвату ресурса одним из них и ожиданию освобождения ресурса другими.

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*.

Понятие сигнала приводится как в инструментальных средствах языка программирования С, так и в стандарте POSIX (*Portable Operating System Interface for uniX*), описывающего соответствующие интерфейсы ядра ОС.
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*;
- *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 специальную 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*, когда генерация его ассоциирована с идентификатором процесса или группы процессов, а также с асинхронным событием, например, с пользовательским вводом с терминала; в каждом процессе определены действия, предпринимаемые в ответ на все предусмотренные системой сигналы;
- *конкретному потоку управления в процессе*, когда сигналы, сгенерированны в результате действий, приписываемых отдельному потоку управления, например, такому как возникновение аппаратной исключительной ситуации.

Говорят, что:

- *сигнал доставлен процессу*, когда *взято для выполнения действие*, соответствующее конкретному процессу и сигналу;
- *сигнал принят процессом*, когда *он выбран и возвращен* одной из функций ***sigwait(...)***.

В интервале, от генерации до доставки или принятия сигнал называется *ждущим*.

Обычно, он невидим для приложений, однако доставку сигнала потоку управления можно блокировать.

Если действие, ассоциированное с заблокированным сигналом, отлично от игнорирования, он будет ждать разблокирования.

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

С сигналом могут быть ассоциированы *действия одного из трех типов*:

- ***SIG_DFL*** — выполняются подразумеваемые действия, зависящие от сигнала, которые описаны в заголовочном файле <***signal.h***>;
- ***SIG_IGN*** - игнорировать сигнал. Доставка сигнала не оказывает воздействия на процесс;
- *указатель на функцию* - обработать сигнал, выполнив при его доставке заданную функцию. После завершения функции обработки процесс возобновляет выполнение с точки прерывания.

Обычно, функция обработки сигнала вызывается в соответствии со следующим заголовком языка С:

```
    void func (int signo);
```

где ***signo*** - номер доставленного сигнала.

Замечание

Первоначально, до входа в функцию ***main(...)***, реакция на все сигналы установлена как *SIG_DFL* или *SIG_IGN*.

Функция называется *асинхронно-сигнально-безопасной* (АСБ), если ее можно вызывать без каких-либо ограничений при обработке сигналов.
В стандарте POSIX-2001, имеется список функций, которые должны быть либо *повторно входимыми*, либо *непрерываемыми сигналами*, что превращает их в АСБ-функции.
В этот список включены порядка 117 функций.
Если сигнал доставляется потоку, а реакция заключается в завершении, остановке или продолжении, весь процесс должен завершиться, остановиться или продолжиться.

Сигнал процессу может быть послан:

- *либо из командной строки* с помощью служебной программы ***kill***;
- *либо из процесса* с помощью одноименной функции:

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

Сигнал задается аргументом ***sig***, значение которого может быть нулевым; в этом случае действия функции *kill(...)* сводятся к проверке допустимости значения **pid**. *Нулевой результат* - признак успешного завершения *kill(...)*.

Если ***pid > 0***, это значение трактуется как идентификатор процесса.
При нулевом значении ***pid***, сигнал посылается *всем процессам из той же группы*, что и вызывающий.
Если значение ***pid*** равно ***-1***, адресатами являются все процессы, которым *вызывающий имеет право посылать сигналы*.
При прочих отрицательных значениях ***pid***, сигнал посылается группе процессов, чей идентификатор равен абсолютной величине **pid**.

Процесс имеет право послать сигнал адресату, заданному аргументом ***pid***, если он (процесс) имеет соответствующие привилегии или его *реальный или действующий UID* совпадает с *реальным или сохраненным UID адресата*.

Вызов служебной программы *kill* в виде: ***kill -l***, позволяет вывести на терминал весь список сигналов:

```
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE    14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT    19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU    23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR     31) SIGSYS      32) SIGRTMIN    33) SIGRTMIN+1
```

```
34) SIGRTMIN+2  35) SIGRTMIN+3  36) SIGRTMIN+4  37) SIGRTMIN+5
38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8  41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2
62) SIGRTMAX-1  63) SIGRTMAX
```

Стандарт языка С определяет имена всего шести сигналов: SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV и SIGTERM.
Стандарт POSIX-2001 определяет как обязательные для реализации сигналы, представленные в таблице 6.3.

Таблица 6.3 - Сигналы, определенные стандартом POSIX

Сигнал	***Описание сигнала***
SIGABRT	Сигнал аварийного завершения процесса. Подразумеваемая реакция предусматривает, помимо аварийного завершения, создание файла с образом памяти процесса.
SIGALRM	Срабатывание будильника. Подразумеваемая реакция — аварийное завершение процесса.
SIGBUS	Ошибка системной шины как следствие обращения к неопределенной области памяти. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти процесса.
SIGCHLD	Завершение, остановка или продолжение порожденного процесса. Подразумеваемая реакция - игнорирование.
SIGCONT	Продолжение процесса, если он был остановлен. Подразумеваемая реакция - продолжение выполнения или игнорирование, если процесс не был остановлен.
SIGFPE	Некорректная арифметическая операция. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGHUP	Сигнал разъединения. Подразумеваемая реакция - аварийное завершение процесса.
SIGILL	Некорректная команда. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGINT	Сигнал прерывания, поступивший с терминала. Подразумеваемая реакция - аварийное завершение процесса.
SIGKILL	Уничтожение процесса (этот сигнал нельзя перехватить для обработки или проигнорировать). Подразумеваемая реакция — аварийное завершение процесса.
SIGPIPE	Попытка записи в канал, из которого никто не читает. Подразумеваемая реакция - аварийное завершение процесса.
SIGQUIT	Сигнал выхода, поступивший с терминала. Подразумеваемая ре-

	акция - аварийное завершение и создание файла с образом памяти процесса.
SIGSEGV	Некорректное обращение к памяти. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти процесса.
SIGSTOP	Остановка выполнения (этот сигнал нельзя перехватить для обработки или проигнорировать). Подразумеваемая реакция — остановка процесса.
SIGTERM	Сигнал терминирования. Подразумеваемая реакция - аварийное завершение процесса.
SIGTSTP	Сигнал остановки, поступивший с терминала. Подразумеваемая реакция - остановка процесса.
SIGTTIN	Попытка чтения из фонового процесса. Подразумеваемая реакция - остановка процесса.
SIGTTOU	Попытка записи из фонового процесса. Подразумеваемая реакция - остановка процесса.
SIGUSR1, SIGUSR2	Определяемые пользователем сигналы. Подразумеваемая реакция - аварийное завершение процесса.
SIGPOLL	Опрашиваемое событие. Подразумеваемая реакция - аварийное завершение процесса.
SIGPROF	Срабатывание таймера профилирования. Подразумеваемая реакция - аварийное завершение процесса.
SIGSYS	Некорректный системный вызов. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти процесса.
SIGTRAP	Попадание в точку трассировки/прерывания. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGURG	Высокоскоростное поступление данных в сокет. Подразумеваемая реакция - игнорирование.
SIGVTALRM	Срабатывание виртуального таймера. Подразумеваемая реакция - аварийное завершение процесса.
SIGXCPU	Исчерпан лимит процессорного времени. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGXFSZ	Превышено ограничение на размер файлов. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.

6.8 Работа пользователя с процессами

Применительно к процессам, пользователь может:

- порождать и завершать процессы;
- просматривать их состояние и изменять приоритет.

Большое прикладное значение в управлениями процессами играют *сигналы*.

Сигнал - это *специальная* и *наиболее экономичная форма взаимодействия* между процессами:

- *сигналы имеют целочисленные номера*, которые различаются процессами;
- *семантика сигнала* должна быть известна процессу для правильного реагирования на его получение;
- *часть сигналов* предназначена *для обработки системой* и не доходит до процесса;
- *часть сигналов* предназначена *для обработки процессом* или игнорирование его.

Новые процессы создаются запуском сценария или исполняемого приложения.

Процессы завершаются:

- *нормально* - в соответствии с работой алгоритма приложения, который исполняется процессом;
- *аварийно* или *нормально* — при получении процессом сигнала;
- *аварийно* — при завершении процесса системой.

Для посылки сигналов процессам, используется утилита **kill**:

```
    kill -сигнал PID...
```

Большинство сигналов предназначено для завершения работы процессов, например,

```
$  kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$
```

где

SIGUP — **Сигнал 1** - Hang UP - повешена трубка; сообщает процессу, что пользователь, запустивший его, вышел из системы. Обычные процессы — завершаются.

SIGINT — **Сигнал 2** — INTerrupt — прерывание; именно этот сигнал посылается, когда пользователь нажимает клавиши Ctrl-C.

SIGQUIT	**Сигнал 3** — QUIT — выход; при нажатии Ctrl-\.
SIGTERM	**Сигнал 15** — TERMinate — прекратить: требует немедленного прекращения работы процесса.
SIGKILL	**Сигнал 9** — KILL — убить: ни одна программа не может перехватить или игнорировать этот сигнал.

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.

В режиме пользователя обычно выполняется множество процессов, значимость которых (*приоритет*) может быть различным и подлежит достаточно сложной настройке.
Пользователь может изменять приоритет выполняемых процессов используя *уровни тактичности*.
Тактичность — целочисленное значение влияющее на количество процессорного времени, выделяемого процессу относительно других процессов.
В Linux, значения тактичности лежат в пределах от -20 до +20:
-20 — максимальный приоритет;
+20 — минимальный приоритет.

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]
```

Замечание
Повышать приоритет процессов может только пользователь **root**.

Для управления процессами также используются команды:
прерывания — нажатие клавиш Ctrl-Z;
bg — перевод процесса в фоновый режим;
fg — перевод процесса в приоритетный режим.

6.9 Системные вызовы ОС по управлению процессами

Изучив управление процессами на макроуровне, что позволяет нам успешно использовать системные и прикладные средства работы с ОС, рассмотрим основные возможности самого процесса влиять на себя и другие процессы.
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 (thred) процесса.

Воздействуя на каждое из свойств процесса, можно управлять им.

Наличие в пользовательском пространсте ОС множества взаимодействующих процессов, каждый из которых может управляться по отдельному свойству, приводит к *большому многообразию возможных системных вызовов* к ядру ОС.
Чтобы упорядочить и, по возможности, минимизировать тупиковые ситуации между поцессами, 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*.

В данном подразделе, мы ограничимся только *функциями планировщика*, обеспечивающими поддержку «жизненного цикла процесса».

Системный вызов fork(…) - обеспечивает порождение нового (дочернего процесса). В случае неудачи, возвращает -1.
Дочерний процесс получает 0.
Родительский процесс получает PID дочернего процесса.
Системный вызов:

```
#include <unistd.h>
pid_t fork(void);
```

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

```
#include <unistd.h>
void _exit(int status);
```

Системный вызов wait(…) - используется родительским процессом для ожидания завершения дочернего процесса и получения его PID.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Системный вызов exec(…) - обеспечивает различные варианты вызовов для модификации процесса.

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
           ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

Замечание

Внешняя переменная (двойной указатель) *environ* обеспечивает доступ к переменным среды пользователя.

6.10 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.

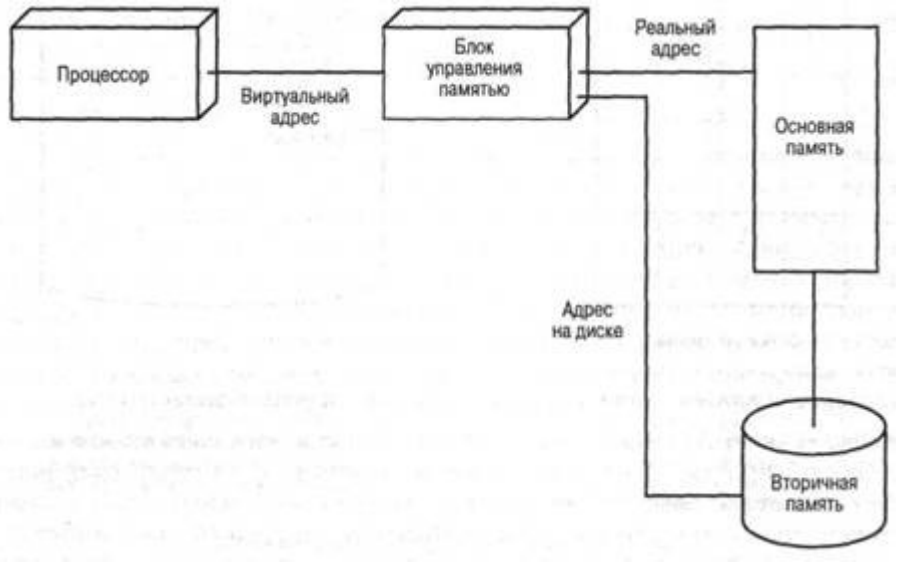

Рисунок 6.8 — Управление памятью ЭВМ

Замечание

В ЭВМ, которые не имеют аппаратной реализации MMU, выполняется эмуляция его работы программой, находящейся в некоторой области ОП.

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.

Относительность физической адресации *ОП* обеспечивается ее *сегментной* или *страничной организацией* и в данной части дисциплины не рассматривается.

Относительность логической адресации *процессора* обеспечивается, например, для процессоров x86, наличием:

- сегментных регистров CS, DS, ES и SS, которые ссылаются на строки дескрипторных таблиц памяти, содержащих *адреса начала сегментов памяти*;
- регистров смещений IP, AX, BX, CX и других, *адресующих ОП относительно начала соответствующих сегментов*

Относительность логической адресации *процесса* обеспечивается, как показано на рисунке 6.9:

- *транслятором исходного текста* программы, создающим объектный код программы с относительной адресацией;
- *перемещающим загрузчиком*, привязывающим части кода и данных процесса к конкретным адресам ОП.

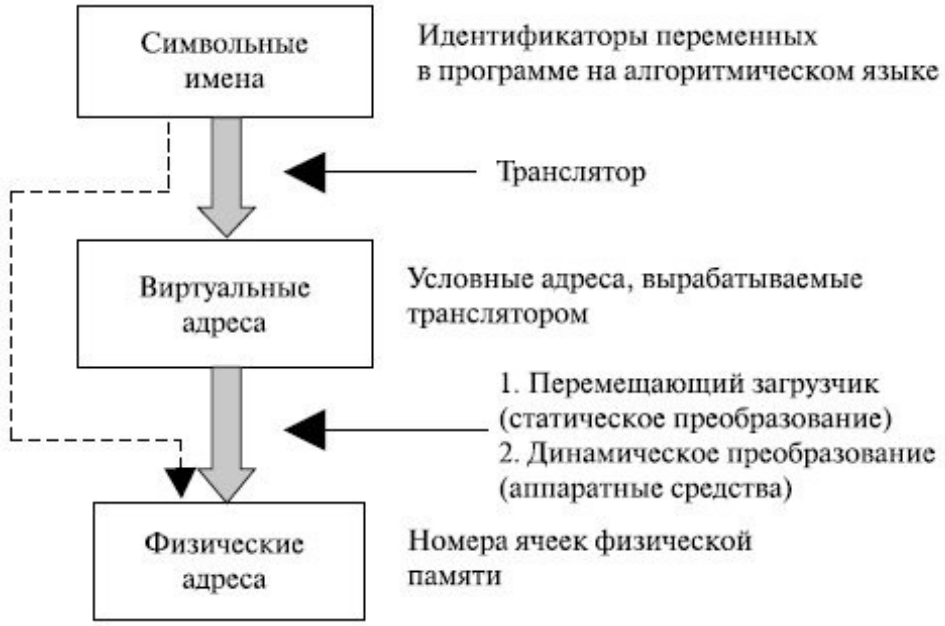

Рисунок 6.9 — Виртуальные адреса процесса

Замечание

Все многобразие способов адресации и размещения процессов в ОП выходит за рамки нашей темы. Как уже было показано на рисунке 6.2, в подсистеме управления процессами выделена специальная подсистема: *распределение памяти*, отвечающая за все эти вопросы и предназначенная для сокрытия всех сложных особенностей работы процесса с памятью.

Активный (*функционирующий*) процесс сам, целиком или частично, находится в ОП, поэтому его возможности по управлению памятью естественным образом ограничены. Чтобы показать, какие возможности у него имеются, рассмотрим обобщенную структуру процесса, на примере ОС Linux.

У каждого процесса, в системе Linux, есть адресное пространство, разделенное на *3 логических сегмента*:

text segment — Содержит машинные команды, образующие исполняемый код программы, который создается компилятором и ассемблером при транс-ляции программы в машинный код (*только для чтения*).

data segment — Содержит переменные, строки, массивы и другие данные программы и состоит из двух частей: *инициализированные данные* и *не-инициализированные данные*.
Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске.
Все переменные, в неинициализированной, части должны быть сброшены в 0.

stack segment — Выделяется отдельный сегмент *на каждую нить* (*thread*) процесса. На большинстве компьютеров он начинается около старших адресов виртуального адресного пространства и *растет вниз к 0*.
Если указатель стека оказывается ниже нижней границы стека, то происходит аппаратное прерывание, при котором операционная система понижает границу сегмента стека на одну страницу.
Когда программа запускается, ее стек не пуст, он *содержит все переменные окружения*, а также *командную строку*, введенную в оболочке (shell) для вызова этой программы.

Если рассмотреть только эти три сегмента, то можно заметить:

- *системные вызовы* fork(...), _exit(...) и exec*(...), косвенно вызывают подсистему управления памятью и изменяют ее виртуальное пространство;
- *сегменты text и stack* недоступны для прямого изменения самому процессу, поскольку напрямую управляются ядром ОС, обеспечивая его существование и функционирование;
- *процессу недоступно прямое управление* первой часть сегмента data, поскольку *эта часть статически сформирована* компилятором языка;
- *процессу доступно прямое управление* только второй частью сегмента data, поскольку *процесс может иметь операторы для динамического формирования данных*.

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

```
#include <stdlib.h>
void *malloc(size_t size); # выделение нужного размера памяти;
void free(void *ptr);      # освобождение выделенного участка памяти;
void *calloc(size_t nmemb, size_t size); # выделение памяти для массива;
void *realloc(void *ptr, size_t size);   # изменение размера выделенной памяти.
```

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

```
#include <unistd.h>
int brk(void *addr);           # установка адреса конца сегмента data
void *sbrk(intptr_t increment); # добавление памяти сегменту data
```

Адреса окончаний сегментов процесса доступны ему через внешние переменные:

etext	Первый адрес, после окончания сегмента кода процесса;
edate	Первый адрес после инициализированных данных сегмента data;
end	Первый адрес после неинициализированных данных сегмента data, известный как BSS segment.

В качестве примера, следующая программа выводит адреса концов своих сегментов:

```
#include <stdio.h>
#include <stdlib.h>

extern char etext, edata, end; /* The symbols must have some type,
                                  or "gcc -Wall" complains */

int
       main(int argc, char *argv[])
       {
           printf("First address past:\n");
           printf("    program text (etext)      %10p\n", &etext);
           printf("    initialized data (edata)  %10p\n", &edata);
           printf("    uninitialized data (end)  %10p\n", &end);

           exit(EXIT_SUCCESS);
       }
```

Конечно, указанных выше средств, даже в совокупности с рассмотренными уже сигналами, недостаточно для эффективного управления в пространстве пользователя сложной конфигурацией процессов, поэтому подсистема управления процессами, показанная на рисунке 6.2, дополняется отдельной *подсистемой взаимодействия процессов*.

Современные ядра ОС содержат множество механизмов для взаимодействия процессов. Одним из первых и базовым является механизм, известный как *набор средств IPC*.
IPC — *InterProcess Communication* — *межпроцесная коммуникация*, включает всебя механизмы: синхронизации, разделения памяти, обмена сообщениями и удаленный вызов процедур (*RPC — Remote Procedure Call*).
Мы, *в данной теме*, рассмотрим только механизмы *разделения памяти* и *обмен сообщениями*.

6.11 Системные вызовы и разделяемая память

Первоначально, все процессы ОС максимально разделены и защищены.
Все процессы обращаются к ядру ОС, требуя, захватывая и потребляя ресурсы.
Ядро ОС планирует запуск процессов на выполнение, обслуживая процессы ресурсами и устраняя возникающие противоречия, когда процессы начинают конкурировать за общий ресурс.
Базовые средства взаимодействия процессов обеспечиваются:

- *средствами поддержки «жизненного цикла процесса»*: порождая, модифицируя и завершая процессы;
- *механизмом сигналов*: распространяя и принимая сигналы, а также обеспечивая корректные действия на их наличие.

Преимущество базовых средств взаимодействия процессов — высокая скорость и эффективность их использования.
Недостатки:

- *необходимость одновременного существования* группы взаимодействующих процессов;
- *слабая информативность сигналов*, требующая хорошего знания их семантики и алгоритмов работы ядра ОС.

Альтернативный подход к взаимодействию процессов - *использование возможностей файловой системы ОС*.
Преимущество: возможность использования любых средств взаимодействия.
Недостатки:

- *алгоритмическая сложность* разрешения всех *тупиковых ситуаций*, возникающих в приложениях;
- *необходимость реализации* средств взаимодействия непосредственно в приложениях;
- *неэффективная и ненадежная реализация* взаимодействия процессов, слабо обеспеченная возможностями ядра ОС;
- *локальная семантика и интерпретация* механизмов взаимодействия ограничивающая переносимость самих приложений.

Комбинированные подходы к взаимодействию процессов:

- *максимальное использование* возможностей ядра ОС;
- *разработка новых эффективных алгоритмов* взаимодействия процессов, которые включаются в функционал ядра ОС.

Одним из таких подходов является проект *IPC* - InterProcess Communication, который предполагает, что:

- *все процессы*, желающие взаимодействовать, обращаются к ядру ОС и получают от него *уникальный ключ*;
- *используя уникальный ключ*, процессы создают в ядре ОС свои ресурсы и потребляют ресурсы, уже созданные другими процессами.

Генерация уникального ключа осуществляется ядром ОС посредством системного вызова:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, int proj_id);
```

где параметры:
pathname - должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию;
proj_id – это небольшое целое число, характеризующее экземпляр средства связи.

Замечание

В случае невозможности генерации ключа функция возвращает отрицательное значение, в противном случае, она возвращает значение сгенерированного ключа. Тип данных *key_t* обычно представляет собой 32-битовое целое число.

Таким образом, для взаимодействия процессов посредством IPC, разработчикам приложений необходимо договориться:

- *об именовании* своего взаимодействия, что определяется двумя параметрами: именем существующего в ОС файла и номера варианта его использования;
- *выбора способа взаимодействия*, поддерживаемого IPC;
- *разработать прикладную часть взаимодействия*, в пределах выбранного способа взаимодействия.

В данном подразделе, мы рассмотрим основную идею способа взаимодействия, основанного на совместном использовании оперативной памяти ЭВМ и получившего название *разделяемой памяти IPC*.

В стандарте POSIX-2001 разделяемый объект памяти определяется как **объект**, представляющий собой память ЭВМ, которая может быть *параллельно отображена* в адресное пространство более чем одного процесса.

Единицей отображения разделяемой памяти являются *сегменты*, в которые процесс может поместить данные любой структуры, читать данные этой структуры и модифицировать их.

Механизм IPC гарантирует процессам, что выделяемый сегмент памяти будет не меньше размера записываемой структуры данных и будет сохранен ядром ОС, даже если процесс завершит свою работу.

Таким образом, вариант взаимодействия с помощью разделяемой памяти обеспечивает процессам:

- *асинхронный способ взаимодействия*, не требующий одновременного существования всех взаимодействующих процессов;
- *наиболее быстрый доступ* к общим структурированным данным.

Непосредственная работа с сегментами разделяемой памяти обеспечивается с помощью четырех системных вызовов:

```
#include <sys/shm.h>

int shmget (key_t key, size_t size, int shmflg);
void *shmat (int shmid, const void *shmaddr, int shmflg);
int shmdt (const void *shmaddr);
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Семантическое назначение этих вызовов следующее:

shmget	Создает разделяемый сегмент, тем самым, специфицируя первоначальные *права доступа* к нему и *его размер в байтах*. Возвращает *идетификатор памяти*, при подключении сегмента к процессу или для управления им.
shmat	Подключает *виртуальный адрес процесса*, указывающий на используемую структуру данных к сегменту памяти с заданным иденитификатором. Возвращает *адрес разделяемого сегмента*.
shmdt	Отключает *адрес разделяемого сегмента* от виртуального адреса процесса.
shmctl	*Управляет разделяемым сегментом* посредством команд и данных специальной структуры. В частности, удаляет разделяемый сегмент.

Замечание

Более подробное изучение системных вызовов, использующих разделяемую па-мять, выходит за рамки данной части нашей дисциплины. Кроме того, следует заметить, все задачи на совместное использование данных подвержены тупиковым ситуациям, которые программисты должны разрешать самостоятельно. Обычно, для этих целей используется механизм семафоров, который также входит в набор средств пакета IPC.

Средства межпроцессного взаимодействия ***System V IPC*** имеют также две специальные утилиты, которые могут использоваться из командной строки или в сценариях языка shell:

- *ipcs* - вывод отчёта о состоянии средств межпроцессного взаимодействия;
- *ipcrm* - удаление очередей сообщений, наборов семафоров и разделяемых сегментов памяти.

Ощий синтаксис этих утилит:

```
ipcs [-abcmopqstMQSTy] [-C дамп] [-N система] [-u пользователь]

ipcrm [-q msqid] [-m shmid] [-s semid] [-Q msgkey] [-M shmkey]
      [-S semkey] ...
```

Для примера, на рисунке 6.10, с пмощью утилиты *ipcs*, выведена информация об используемых ОС сегментах разделяемой памяти

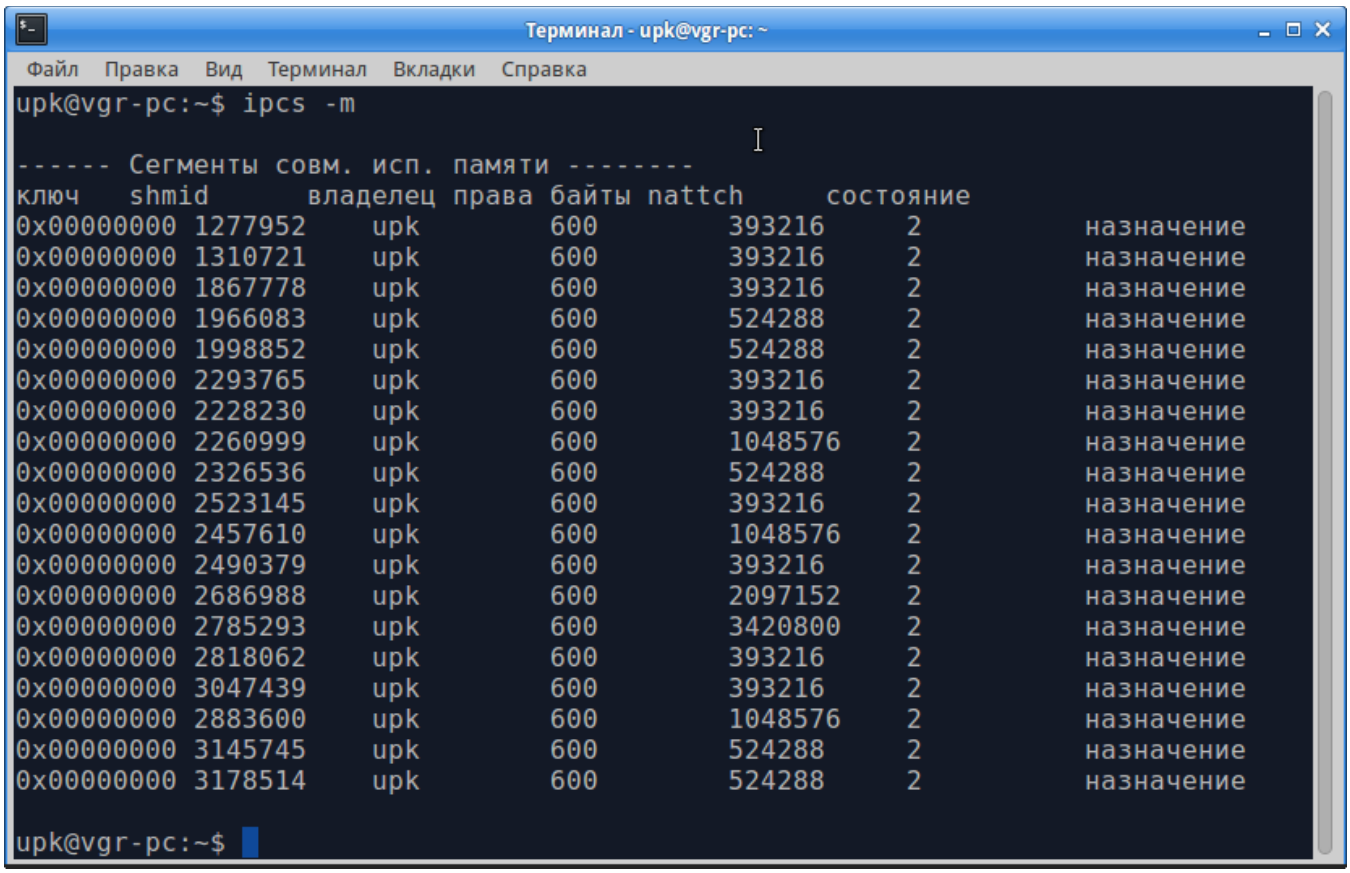

Рисунок 6.10 — Используемые ОС сегменты разделяемой памяти

6.12 Передача сообщений

Как уже было отмечено ранее, *основной недостаток* взаимодействия процессов с помощью сигналов является их малая информативность и специфическая семантика, ориентированная в первую очередь на потребности ядра ОС.
Процессам, функционирующим в пользовательском режиме ОС, необходимы свои средства сигнализации, которые бы позволяли:

- *увеличить информативность сигналов*, наполнив их семантикой прикладного содержания;
- *обеспечить асинхронное распространение сигналов*, не требующее совместного существования взаимодействующих процессов.

Траким средством, входящим в *System V IPC*, являются *очереди сообщений*.

В стандарте POSIX-2001, *очереди сообщений* — набор типизированных объектов, которыми процессы могут асинхронно взаимодействовать между собой, не затрачивая свои прикладные ресурсы на организацию и сопровождение самих очередей.
Очереди сообщений - это наиболее семантически нагруженный способ взаимодействия процессов через линии 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 одну и ту же линию связи для передачи данных *в двух направлениях между несколькими процессами*.

Общая структура сообщения описана в файле *<sys/msg.h>* , как ***struct msgbuf***:

```
struct msgbuf {
     long mtype;     # строго положительная величина, задающая тип сообщения
     char mtext[1]; # адрес начала самого сообщения
};
```

Следуя этому шаблону, разработчик приложений может формировать и использовать свою собственную семантику приложений, например:

```
struct mymsgbuf {
     long mtype;
     char mtext[1024];
} mybuf;
```

или

```
struct mymsgbuf {
     long mtype;
     struct {
          int iinfo;
          float finfo;
     } info;
} mybuf;
```

Замечание

Максимальный размер сообщения ограничен по разному для разных ОС. Например, в ОС Linux, максимальный размер сообщения равен 4080 байт. Администратор ОС дополнительно может уменьшить это ограничение.

Следуя общим правилам System V IPC, создание и доступ к очередям сообщений осуществляется *посредством ключа*, который генерируется с использованием системного вызова *ftok(...)*, рассмотренного в предыдущем подразделе.
Непосредственная работа с очередями сообщений обеспечивается с помощью четырех системных вызовов:

```
#include <types.h>
#include <ipc.h>
#include <msg.h>

int msgget(key_t key, int msgflg);
int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
int msgrcv(int msqid, struct msgbuf *ptr, int length, long type, int flag);
int msgctl(int msqid, int cmd,
           struct msqid_ds *buf);
```

Не вдаваясь в детали практического использования этих системных вызовов, кратко опишем их основное семантическое назначение:

msgget	Создает очередь сообщений или подключается к уже созданной очереди, используя сгенерированный функцией *ftok(...)* *уникальный ключ*. Возвращает *целочисленный дескриптор очереди*, который используется остальными тремя системными вызовами.
msgsend	Помещает в очередь сообщение заданного типа.
msgrcv	Читает из очереди сообщение заданного типа.
msgctl	Управляет очередью с использованием специальной структуры типа *msgid_ds*. В частности, удаляет очередь из системы.

Замечание

Механизм очередей сообщений имеет достаточно развитый инструментарий их использования, который поддерживается ядром ОС в соответствии с общими правилами использования разделяемых ресурсов:

- поддерживается контроль прав доступа;
- блокирование читающего процесса, если в очереди отсутсвует сообщение с заданным типом;
- имеются различные варианты порядка чтения сообщений из очереди;
- имеются развитые средства выявления и обработки ошибочных ситуаций.

В целом, механизм очередей сообщений и другие средства System V IPC являются достаточно низкоуровневыми средствами взамодействия с ОС. Это требует от разработчиков системных и прикладных программ хорошей профессиональной подготовки, поэтому для прикладных целей используются более высокоуровневые средства, например, *шина D-Bus* и другие.

В частности, для анализа и администрирования очередей сообщений могут использоваться, уже уромянутые в предыдущем подразделе, утилиты: *ipcs* и *ipcrm*.

6.13 Лабораторная работа по теме №6

Цель лабораторной работы №6 — практическое закрепление учебного материала по теме «Управление процессами ОС».
Метод достижения указанной цели — закрепление учебного материала, изложенного в первом разделе пособия посредством утилит ОС, а также выполнение заданий, приведенных в данном разделе.
Чтобы успешно выполнить данную работу, студенту следует:

- *запустить с flashUSB* ОС УПК АСУ, подключить личный архив и переключиться в сеанс пользователя ***upk***;
- *запустить на чтение* данное пособие и на редактирование личный отчет;
- *открыть одно или несколько окон терминалов*, причем хотя бы в одном окне терминала открыть Midnight Commander, для удобства работы с файловой системой ОС;
- *приступить к выполнению работы*, последовательно пользуясь рекомендациями представленных ниже подразделов.

Замечание

Многие команды ОС студенту еще не известны, поэтому следует:

- для вывода на консоль руководства по интересуемой команде, использовать: ***man имя_команды***;
- для выяснения сущесвования команды, ее доступности и местоположения, использовать: ***command -v имя_команды***;
- для уточнения правил запуска конкретной команды, можно попробовать один из вариантов: ***команда --help*** или ***команда -h*** или ***команда -?***.

В процессе выполнения лабораторной работы студент заполняет личный отчет по каждому изученному вопросу!

6.13.1 Сценарий загрузки ОС

Прочитайте и усвойте учебный материал подразделов 6.1 и 6.2.
Запустите Midnight Commander и бедитесь в наличии файлов: */bin/sh* и */sbin/init*.
Запустите файловый менеджер thunar, перейдите в директорию */etc/upkasu* и запустите на просмотр с помощью ***mousepad*** сценарий *init*, который по содержанию соответствует файлу *init*, размещенному в корне временной файловой системы ОС.
Изучите и опишите содержимое этого файла.
Особое внимание обратите на:

- певоначальное создание устройств;
- чтение параметров, передаваемых ядру ОС с помощью ПО GRUB;
- монтирование и перенос файловых систем;
- запуск основного процесса *init*.

В случае затруднений, обращайтесь к преподавателю за консультацией!

6.13.2 Разные подходы к управлению процессами

Прочитайте и усвойте учебный материал подразделов 6.3 - 6.6.
Изучите назначение и работу утилит ***runlevel***, ***telinit*** и ***initctl***, а также содержимое директорий */etc/init* и */etc/init.d*.
Опишите основные отличия подходов **SysVinit** и **upstart**.
С помощью руководства **man**, изучите *systemd* и *systemctl*, а также изучите и опишите содержимое директорий */lib/systemd* и */etc/systemd*.

6.13.3 Сигналы и средства IPC

Прочитайте и усвойте учебный материал подразделов 6.7 - 6.12.
Изучите и освойте работу с утилитами ***kill***, ***ipcs*** и ***ipcrm***.
Сделайте краткое заключение по всем лабораторным работам: отметьте какие вопросы изложены недостаточно полно и требуют более глубокого изучения.

Заключение

Завершив курс обучения, изложенный в данном методическом пособии, студент получает два уровня знаний:

- обобщенный уровень представления о предмете, связанный с изучением первых трех тем, обеспечивающий основные архитектурные представления об операционных системах, их методах загрузки и языку командного управления ОС (shell, - Bourne Shell);
- конкретизированный уровень представления о предмете, связанный с изучением последних трех тем, обеспечивающий должное представление о файловых системах, пользователях и процессах.

Закрепив полученные знания выполением лабораторных работ по каждой теме данного курса, оформив общий отчет о проделанной работе и защитив единый отчет в процессе индивидуального общения с преподавателем, студент считается готов к проведению следующей аттестации:

- зачету - для направления подготовки: 09.03.01 - Информатика и вычислительная техника;
- экзамену - для направления подготовки: 09.03.03 - Прикладная информатика.

В целом, учебный материал данного учебно-методического пособия дает только начальные представления об операционных системах, достаточные для подготовки специалистов экономических специальностей, ориентированных на прикладную часть использования ОС. Для специальностей, ориентированных на разработку программного обеспечения, изученных знаний - недостаточно, особенно в части информации о системах разработки и языке системного программирования С. Это — предмет второй части изучаемой дисциплины, ориентированной на прикладной языковый образовательный уровень специалиста, соответствующий направлению подготовки 09.03.01.

Список использованных источников

1 Таненбаум Э. Современные операционные системы. - СПб.: Питер, 2012. - 1020 с.
2 Синицын С.В. Операционные системы: учебник для вузов. - М.: Академия, 2012. - 304 с.
3 Резник В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 33 с.

Учебное издание

Резник Виталий Григорьевич

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебно-методическое пособие предназначено для изучения теоретических вопросов и выполнения лабораторных работ по дисциплине «Операционные системы» для студентов кафедры АСУ ТУСУР уровня основной образовательной программы бакалавриат направлений подготовки: «09.03.01 - Информатика и вычислительная техника» и «09.03.03 - Прикладная информатика».

Учебно-методическое пособие

Усл. печ. л. 21,14. Тираж . Заказ .
Томский государственный университет
систем управления и радиоэлектроники
634050, г. Томск, пр. Ленина, 40