МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Факультет математики и информационных технологий
Кафедра «Информационных технологий»

Чичев А.А., Чекал Е.Г.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть 1. РАБОТА С ОПЕРАЦИОННОЙ СИСТЕМОЙ

Учебно-методическое пособие

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Факультет математики и информационных технологий
Кафедра «Информационных технологий»

Чичев А.А., Чекал Е.Г.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть 1. РАБОТА С ОПЕРАЦИОННОЙ СИСТЕМОЙ

Учебно-методическое пособие

Ульяновск
2015

УДК 004.052
ББК 32.973
Ч 78

Печатается по решению Ученого совета
факультета математики, информационных и авиационных технологий
Ульяновского государственного университета
(протокол № от)

Рецензенты:
к.т.н., доцент, д.п.н., профессор, заведующий кафедрой «Информатика» УлГПУ
Шубович В.Г.
заместитель директора ОГБУ «Электронный Ульяновск»
Правительства Ульяновской области Клочков А.Е.

Чичев А.А.
Ч78 **Операционные системы**. Часть 1. Работа с операционной системой. Учебно-методическое пособие. / Чичев А.А., Чекал Е.Г. – Ульяновск: УлГУ, 2015. – с.

Учебно-методическое пособие составлено в соответствии с программой дисциплины «Операционные системы», и предусматривает подготовку инженеров и бакалавров по направлениям 09.03.02 «Информационные системы и технологии», 09.03.03 «Прикладная информатика», 02.03.03 «Математическое обеспечение и администрирование информационных систем», 11.03.02 «Инфокоммуникационные технологии и системы» и специальности 10.05.01 «Компьютерная безопасность». Может использоваться студентами родственных специальностей и направлений.

Пособие состоит из трех частей. В части первой приведены краткие сведения о работе основных подсистем операционных систем, а также методические указания к лабораторным работам по установке, конфигурированию и эксплуатации операционных систем. Во второй части пособия рассматриваются устройство средств хранения, форматы разбиения, файловые системы: ufs/ufs2, клоны ufs (в т.ч. ext), fat12/16/32, ntfs, iso 9660. В третьей части пособия рассматриваются ЭМВОС, сетевые технологии и стандарты на них, стеки сетевых протоколов (SMB, IPX/SPX, TSP/IP, AppleTalk, SNA), именование сетевых объектов на различных уровнях ЭМВОС, взаимодействие процессов и сервисы.

Пособие предназначено для практического руководства при проведении преподавателями лабораторных занятий и выполнении заданий студентами указанных направлений и специальностей всех форм обучения.

УДК 004.052
ББК 32.973

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Исторически сложилось так, что в настоящее время самой быстро развивающейся, самой функционально продвинутой, самой сложной алгоритмически, самой большой по объёму и вообще самой-самой операционной системой стала операционная система Linux. Возможно, кому-то это высказывание не нравится, но, увы, оно истинно.

Именно в развитие этой операционной системы вкладываются усилия тысяч высококвалифицированных разработчиков, как частных лиц, так и сотрудников фирм. Объём проекта (Linux-3.x) превысил 16 (шестнадцать!) млн. строк кода — ещё никогда человечество не разрабатывало столь сложные и объёмные программные системы.

Именно эта операционная система обеспечивает функционирование самых мощных ЭВМ нашего мира — согласно списка Top-500 её используют более 90% суперЭВМ [5].

Именно над развитием этой операционной системы работают сотрудники крупнейших корпораций мира — IBM, HP, Oracle, Intel, Google, Samsung и других, и даже Microsoft (!). Ещё совсем недавно, в 90-ые годы, основной вклад в развитие Linux делали частные лица. Однако, в последнее десятилетие в крупнейших ИТ-корпорациях мира появились подразделения, специализирующиеся на разработке в Linux и на её совершенствовании.

Именно эта операционная система совместно с другими Unix'ами является основой Интернета, а «Интернет — это наше всё».

Именно эта операционная система совместно с другими Unix'ами и Unix-подобными ОС является базой для построения высоконадёжных высокодоступных (24x7) информационных систем.

Именно эта операционная система наряду с другими Unix'ами используется на серверах корпораций, банков, бирж, является основой систем управления в энергетике (в том числе АЭС), на транспорте (диспетчерские системы), в связи (АТС, провайдеры Интернета и сотовой связи), в государственных структурах.

То есть, зависимость современной экономики от Unix'овых операционных систем (в том числе, Linux) не просто большая, а основополагающая: если вдруг завтра проснёмся, а Unix/Linux исчез (вот только вчера был — и нет его . . .), то мало не покажется — в 2008 году только один банк «лопнул» и это привело к мировому экономическому кризису, а если все и всё?

Поэтому в данном пособии рассматривается в основном операционная система Unix (и прежде всего — Linux), а остальные упоминаются по мере необходимости в целях сравнения.

Пособие состоит из трёх частей. В первой части приведены краткие сведения о работе основных подсистем операционных систем, а также методические указания к лабораторным работам по установке, конфигурированию и эксплуатации операционных систем.

В первом разделе описываются:

- общие сведения о дисциплине «Операционные системы», о её месте в квалификации специалистов, о роли и значении операционных систем для экономики и общественной среды в целом и конкретных специалистов, в частности,

- определение, состав и структура операционной системы, внешние и внутренние интерфейсы, определено понятие процесса в операционной системе, описаны основные алгоритмы управления процессами и методы взаимодействия процессов,

- определены различные виды памяти вычислительных систем и способы управления памятью, файловые системы на устройствах долговременной памяти и особенности подсистемы ввода/вывода операционной системы,

- понятие оболочки операционной системы и операционной среды,

- введение в системное программирование.

В втором разделе представлены:

- методические указания к лабораторным работам по установке, конфигурированию и эксплуатации операционных систем.

Во второй части пособия рассматриваются устройство средств хранения, форматы разбиения, файловые системы: ufs/ufs2, клоны ufs (в т.ч. ext), fat12/16/32, ntfs, iso 9660.

В третьей части пособия рассматриваются ЭМВОС, сетевые технологии и стандарты на них, стеки сетевых протоколов (SMB, IPX/SPX, TSP/IP, AppleTalk, SNA), именование сетевых объектов на различных уровнях ЭМВОС, взаимодействие процессов и сервисы.

Пособие предназначено для практического руководства при проведении преподавателями лабораторных занятий и выполнении заданий студентами указанных специальностей всех форм обучения.

Учебно-методическое пособие составлено в соответствии с программой дисциплины «Операционные системы», и предусматривает подготовку инженеров и бакалавров по направлениям 09.03.02 «Информационные системы и технологии», 09.03.03 «Прикладная информатика», 02.03.03 «Математическое обеспечение и администрирование информационных систем», 11.03.02 «Инфокоммуникационные технологии и системы» и специальности 10.05.01 «Компьютерная безопасность». Может использоваться студентами родственных специальностей и направлений.

1. ОСНОВНЫЕ ПОНЯТИЯ ОПЕРАЦИОННЫХ СИСТЕМ

1.1. Общие сведения о вычислительных системах

1.1.1. Виды вычислительных систем

ПЭВМ. Самым распространённым видом вычислительных систем до недавнего времени были ПЭВМ (Персональная ЭВМ: desktop, ноутбук, нетбук, iPad, iFon или просто смартфон). В ПЭВМ всё техническое обеспечение делится на две группы: встроенное (системная плата + CPU + память (внутренняя) + светодиоды и переключатели (джамперы) на системной плате) и внешнее — всё остальное, подключаемое к системной плате некоторым образом через разнообразные разъёмы.

Встраиваемые ЭВМ. Достаточно широко распространены встраиваемые ЭВМ (embedded, blade, ТЭЗ (Типовой Элемент Замены — старое советское название встраиваемой ЭВМ)). Они аналогичны ПЭВМ, в них всё техническое обеспечение делится на две такие же группы — встроенное и внешнее с небольшой разницей: набор внешнего подключаемого оборудования для каждого вида встроенной ЭВМ может быть очень небольшим, иногда ограничивается только датчиками, кнопками и лампочками (светодиодами). Пример: встраиваемые ЭВМ стиральных машин, холодильников, мультиварок, автомобилей, самолётов, автоматических межпланетных станций и другой бытовой и небытовой техники. Нередко в сложных технических устройствах устанавливается несколько встроенных ЭВМ и они объединяются сетью в единое вычислительное пространство. Например, в истребителе СУ-34 более тридцати встроенных ЭВМ, совместно обеспечивающих функционирование истребителя.

СуперЭВМ. Менее распространены СуперЭВМ. Их существует два вида: собранные из обычных ПЭВМ или встраиваемых ЭВМ — кластеры, а также оригинальные суперЭВМ («классические»), собранные из компонентов, специально разработанных для каждой конкретной суперЭВМ. В последнее десятилетие появился новый вид суперЭВМ — распределённые суперЭВМ на основе grid-сетей или «облаков».

1.1.2. Обеспечения вычислительных систем

Любая вычислительная система (или просто ЭВМ) состоит из двух «обеспечений» (вообще-то, не только из этих двух, но остальные здесь рассматриваться не будут):

1) технического обеспечения:

- это процессор (CPU),

- память различного вида (внутренняя — несколько видов: оперативная, кэш, flash (не устройство flash-диск (флэшка), а именно внутренняя память на микросхемах flash-памяти,

припаянных к системной плате), внешняя — несколько видов: HDD, SDD, flash-диски, DVD-ROM);

- средства отображения информации: на современных ЭВМ это чаще всего графическая карта с подключенным к ней монитором, ранее использовалась электрическая пишущая машинка, или просто лампочки (светодиоды);

- средства ввода информации: клавиатура, мышка или ШРУ (Шаровое Ручное Устройство), джойстик, тумблер (или просто кнопка), различные датчики (оптические, температурные, давления, влажности, движения, гравитационные и прочие);

- разнообразные внешние устройства очень разного назначения — принтеры, сканеры, видео-камеры и другие устройства;

2) программного обеспечения:

- операционная система;

- системное программное обеспечение: это программы, способствующие функционированию вычислительной системы (утилиты, программы мониторинга и управления вычислительной системой) и программы, способствующие разработке прикладных программ;

- прикладное программное обеспечение: прежде всего это различные business программы (программное обеспечение поддержки экономической деятельности: различные ИС (Информационные Системы), офисные программы, специализированные программы обработки данных) — это самый важный и дорогой вид программного обеспечения, также это игры, научное программное обеспечение (управление научными экспериментами, вычислительное и аналитическое программное обеспечение), программы поддержки образовательного процесса и другие прикладные программы.

1.2. Определение операционной системы

Операционная система это программа, которая выполняет функции управления вычислениями (вычислительными процессами) в компьютере, распределяет ресурсы вычислительной системы между различными вычислительными процессами и образует ту программную среду, в которой выполняются прикладные программы пользователей.

Что важного в этом определении.

1. Операционная система — это программа. Это единственная программа, которая выполняется на ЭВМ с включения ЭВМ и до выключения. Все остальные программы включаются на ЭВМ с помощью операционной системы, выполняются, обращаясь за ресурсами к операционной системе, управляются операционной системой или человеком (оператором, пользователем) с помощью операционной системы, выключаются операционной системой или с её помощью.

2. Когда пользователь запускает некоторую программу на выполнение, то тем самым он (пользователь) даёт команду операционной системе, указывая имя программы (исполняемого файла — файла, в котором в специальном формате (в Unix - в формате ELF - Executable and Linkable Format) хранятся коды (команды) программы и описания структур данных, которые программа должна обрабатывать. Операционная система, на основе информации из этого файла создаёт объект — процесс. То есть, процесс — это запущенная на исполнение программа. Обратите внимание, что здесь **под термином «процесс» понимается объект, а не деятельность.**

3. На современных вычислительных системах одновременно выполняются десятки и сотни процессов. Некоторые процессы — системные, то есть, необходимые для обеспечения функционирования вычислительной системы и их, как правило, включает сама операционная система), другие процессы — запущены пользователями, это обычно прикладные процессы. Поскольку процессов много, то операционная система должна каким-то образом уметь управлять этими процессами. Процессы для своей работы требуют ресурсов, но ресурсов «всегда не хватает чуть-чуть», и почти все из них (ресурсов) существуют в единственном экземпляре (принтер, сканер, процессор, таймер, винчестер, различные сервисы и т. д.). Следовательно, операционная система должна выполнять важные функции:

- уметь управлять процессами: включать, приостанавливать (прерывать), восстанавливать выполнение, завершать (выключать),

- уметь распределять ресурсы между процессами и контролировать использование ресурсов.

4. Современные вычислительные системы, как правило, допускают совместную работу нескольких пользователей. Следовательно, операционная система должна уметь разграничивать деятельность пользователей: обеспечивать сохранность информации пользователей, предоставляя только контролируемый доступ к чужой информации, не позволять пользователям вмешиваться в работу других пользователей (в работу программ, запущенных другими пользователями), не позволять несанкционированное использование вычислительной системы.

Отсюда следует ещё одна важная функция операционной системы: организация безопасной работы пользователей и программ посредством ограничения доступа в систему пользователей и защиты их друг от друга.

Примечание. Но пользователь, работая на вычислительной системе, всегда работает на ней с помощью каких-либо программ. Следовательно, говоря о разграничении и защите пользователей, мы всегда говорим о разграничении и защите процессов друг от друга.

5. За долгую историю вычислительной техники были созданы самые разнообразные вычислительные системы, которыми мы сейчас пользуемся. Для их создания использовались разные идеи, разные концепции. Соответственно, вычислительные системы имеют разную «архитектуру»: построены на разных процессорах, разных системных платах, разных

микросхемах, используют разное оборудование, разные принципы именования элементов.

В настоящее время используются десятки различных архитектур вычислительных систем. И нередко у пользователей возникает необходимость запуска одних и тех же программ на этих разных вычислительных системах. Соответственно, программисты должны уметь создавать программы для вычислительных систем, имеющих разную архитектуру. Очевидно, что средний программист не в состоянии учитывать все особенности различных архитектур и различного оборудования. Следовательно, операционная система должна выполнять ещё одну важную функцию: предоставлять программисту и пользователю некоторую стандартизированую «абстрактную» виртуальную машину, независимую от того, на каком оборудовании операционная система работает, и эта «абстрактная» машина должна скрывать внутри себя все особенности конкретных архитектур и оборудования.

6. Таким образом, операционная система — это единственная программа, которая выполняется на вычислительной системе с момента включения (вычислительной системы) и до её выключения и которая выполняет следующие основные функции:

1) предоставляет процессам «абстрактную» вычислительную машину,

2) управляет ресурсами (доступом к ресурсам),

3) организует многозадачную среду и разграничивает процессы друг от друга,

3') организует многопользовательскую среду и разграничивает пользователей друг от друга,

4) учитывает процессы, ресурсы и пользователей.

С учётом примечания к подпункту 4, функции 3 и 3' — являются по сути одним и тем же.

Следовательно, ОС — это единственная программа на ЭВМ, которая выполняется с момента включения ЭВМ и до выключения и которая выполняет четыре основных функций, перечисленных выше.

1.3. Структура и архитектура операционной системы

1.3.1. Место операционной системы

На рисунке 1 показано место операционной системы в вычислительной системе.

Подобное разбиение вычислительной системы соответствует, скажем так, «слоёному» типу построения вычислительных систем [1]: каждый слой (подсистема) может взаимодействовать лишь с соседними слоями. Кроме того, зависимости между подсистемами организованы по принципу «сверху вниз» («интерфейсный» тип): слои, изображённые ближе к вершине, зависят от нижних слоёв, в то время как расположенные ближе к основанию не зависят от верхних.

Рис. 1. Состав вычислительной системы

1.3.2. Структура операционной системы

Ядро Linux состоит из пяти основных подсистем [2]: планировщик процессов, менеджер

памяти, виртуальная файловая система, сетевой интерфейс, подсистема межпроцессного взаимодействия.

Планировщик процессов (process scheduler) предназначен для обеспечения доступа процессов к центральному процессору. Планировщик реализует политику (стратегию), гарантирующую, что процессы будут иметь доступ к процессору.

Менеджер памяти (memory manager) обеспечивает процессам безопасный доступ к общей памяти компьютера. Кроме этого, менеджер памяти поддерживает виртуальную память, позволяя операционной системе работать с процессами, использующими памяти больше, нежели её есть в системе («физической» памяти). Содержимое памяти, неиспользуемое в данный момент времени, перемещается (swapped out) в постоянное хранилище, в специальный swap-раздел жесткого диска с использованием файловой системы swapfs, чтобы потом в случае необходимости вернуть (swapped back) обратно в оперативную память.

Виртуальная файловая система (virtual file system, VFS) абстрагирует детали различных аппаратных устройств, предоставляя общий файловый интерфейс к этим устройствам. Кроме этого, виртуальная файловая система поддерживает несколько форматов файловых систем, совместимых с другими операционными системами.

Сетевой интерфейс (network interface) предоставляет доступ к стеку сетевых протоколов, и к различному сетевому оборудованию.

Подсистема межпроцессного взаимодействия (inter-process communication, IPC) поддерживает несколько механизмов взаимодействия между процессами внутри одной операционной системы.

На рисунке 2 показано высокоуровневое разбиение ядра на подсистемы, где стрелками показана взаимозависимость подсистем ядра. На рисунке видно, что наиболее важной, центральной подсистемой ядра является планировщик процессов: все другие подсистемы зависят от него, потому что всем им нужно приостанавливать и возобновлять процессы.

Обычно подсистема будет держать процесс в приостановленном состоянии, если он ждет завершения некой (инициированной им) операции с оборудованием или сервисом, и возобновлять процесс по завершении данной операции. Например, когда процесс пытается послать сообщение по сети, сетевому интерфейсу может понадобиться приостановить этот процесс до тех пор, пока сетевое оборудование не отправит данное сообщение. После того, как оно будет отправлено (или оборудование вернет сообщение об ошибке), сетевой интерфейс возобновит приостановленный процесс и передаст ему код возврата, показывающий, была ли операция завершена успешно или нет.

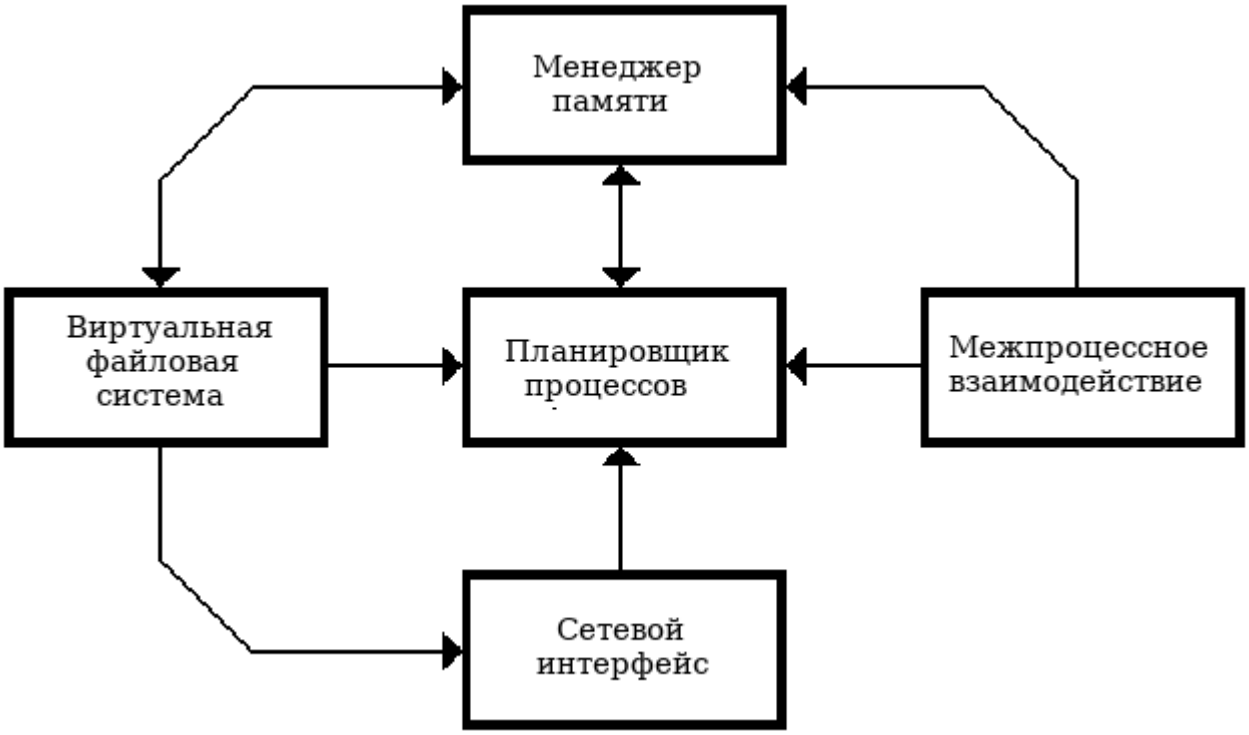


Рис. 2. Подсистемы ядра операционной системы Linux

Аналогично, все прочие подсистемы (менеджер памяти, виртуальная файловая система и межпроцессное взаимодействие) зависят от планировщика процессов в силу схожих причин.

Другие зависимости чуть менее очевидны, но столь же важны:

1. Планировщик процессов использует менеджер памяти для настройки схемы распределения аппаратной памяти для определенного процесса, когда процесс этот возобновлен.

2. Подсистема межпроцессного взаимодействия использует менеджер памяти, чтобы поддерживать механизм связи с совместно используемой памятью. Этот механизм позволяет двум процессам иметь доступ к некой общей области памяти (не считая того, что каждый из процессов имеет и собственную память).

3. Виртуальная файловая система использует сетевой интерфейс для поддержки сетевой файловой системы (NFS), а также использует менеджер памяти для предоставления такого устройства, как RAM-диск.

4. Менеджер памяти использует виртуальную файловую систему для поддержки подкачки (swapping) или перемещения страниц (paging); это единственная причина, по которой менеджер памяти зависит от планировщика процессов. Когда процесс хочет получить доступ к содержимому памяти, которое в данный момент времени сохранено (swapped out) в некотором постоянном хранилище (как правило, в специальном swap-разделе жесткого диска), менеджер памяти обращается с запросом к файловой системе достать это содержимое из хранилища и приостанавливает процесс, пока запрос не будет завершён.

В дополнение к уже рассмотренным зависимостям, все подсистемы в ядре полагаются на

некие общие ресурсы, не показанные на рисунке. Ими являются процедуры, используемые подсистемами ядра для выделения свободной памяти (для нужд ядра), процедуры для печати предупреждений и сообщений об ошибках, а также системные отладочные процедуры.

Каждая из изображённых на рисунке 2 подсистем содержит информацию о своём состоянии, и информация эта доступна другим подсистемам путём использования процедурного интерфейса (межмодульного); более того, каждая из подсистем несёт ответственность за сохранение целостности обслуживаемых ею ресурсов.

На рисунке 3 структура ядра показана более детально [3]. К сожалению, размер рисунка, тем более черно-белого, не позволяет передать всю красоту структуры ядра. Однако, даже на таком рисунке видна «слоистость» ядра: чётко заметно внутреннее ядро операционной системы и несколько слоёв модулей. Модули каждого слоя обращаются к модулям нижестоящего слоя, сами, в свою очередь, предоставляя модулям вышестоящего слоя интерфейс.

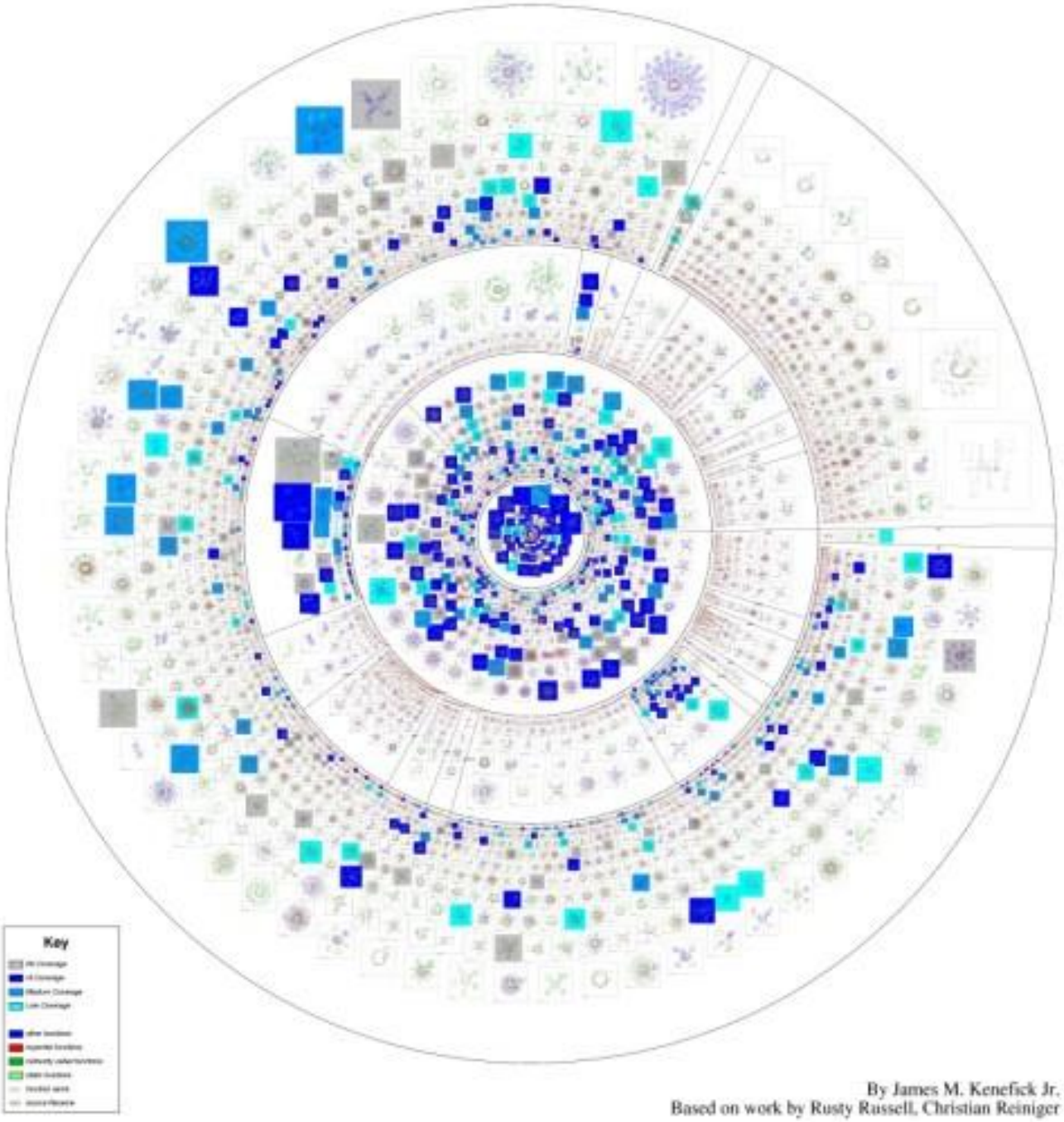


Рис. 3. Структура ядра операционной системы Linux

1.3.3. Архитектуры операционных систем

Монолитная система. Операционная система — это почти обычная программа, поэтому и организовано оно так же, как устроено большинство программ, то есть, состоит из процедур и функций. Операционные системы Unix/Linux написаны на языке Си, следовательно, состоят из функций. То есть, компоненты операционной системы являются не самостоятельными модулями (программами), а составными частями одной большой программы. Такая структура операционной системы называется *монолитным ядром (monolithic kernel)*.

Монолитное ядро представляет собой набор функций, каждая из которых может вызвать каждую. Все функции работают в привилегированном режиме. Таким образом, монолитное ядро это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова функций. Для монолитной операционной системы ядро операционной системы совпадает со всей операционной системой.

Во многих операционных системах с монолитным ядром сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается операционная система. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонент повышает надежность операционной системы в целом.

Монолитное ядро старейший способ организации операционных систем. Примером систем с монолитным ядром является большинство Unix-систем, в частности операционные системы класса opensource (FreeBSD, Linux) могут быть приведены к монолитному виду.

Слоёная система. Более сложную структуру имеют «слоёные» системы (*layered systems*), когда вся вычислительная система делится на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты из уровня N-1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем - интерфейс пользователя (интерфейс прикладного программирования - API). Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне.

Слоеные системы хорошо реализуются. При использовании операций нижестоящего слоя не нужно знать, как они реализованы, нужно знать лишь, что они делают (то есть, интерфейс нижестоящего слоя).

Слоеные системы хорошо тестируются. Отладка начинается с нижнего слоя и проводится послойно. При возникновении ошибки мы можем быть уверены, что она находится в тестируемом слое. Слоеные системы хорошо модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложнее для разработки: предварительно нужно правильно определить порядок слоев, и какая функция к какому слою относится.

Слоеные системы менее эффективны, чем монолитные. Так, например, для выполнения операций ввода-вывода в программе пользователя системному вызову придется последовательно пройти все слои - от верхнего до нижнего.

Примером слоёной системы является операционная система minix, разработанная Э. С. Таненбаумом со студентами в учебных целях.

Микроядерная система. Возможна разработка операционных систем с перенесением значительной части системного кода на уровень пользователя и одновременная минимизация ядра. Такой подход к построению ядра называется микроядерной архитектурой (*microkernel architecture*) операционной системы (рис. 4).

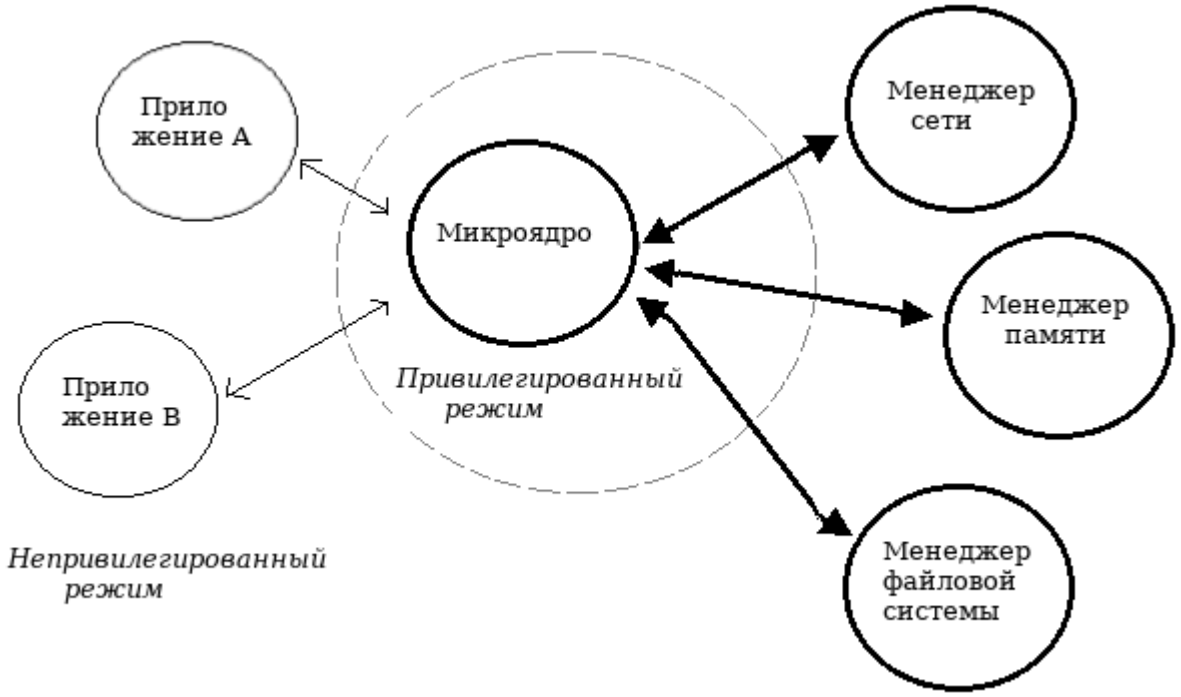


Рис. 4. Микроядерная архитектура.

В этом случае большинство ее составляющих являются самостоятельными программами и взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью. Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

Преимуществом микроядерной архитектуры является высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонент. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонент ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства.

В то же время, микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с передачей сообщений между модулями, что существенно влияет на производительность. Для того чтобы микроядерная операционная система по скорости не

уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем это необходимость очень аккуратного проектирования.

Примеры микроядерных систем: VxWorks, QNX, minix-3.

Смешанные (гибридные) системы. Это современная тенденция в разработке универсальных операционных систем (предназначенных для широкого использования). Примером этого подхода является операционная система Linux, что наглядно демонстрирует рисунок 3. Она имеет монолитно-слоёно-модульную архитектуру. То есть, наряду с существованием в ней монолитного слоёного ядра, включающего основные модули системы (см. рис.2) - «Управление процессами», «Управление памятью», «Виртуальная файловая система», «Подсистема ввода-вывода», «Сетевая подсистема», «Межпроцессное взаимодействие», в ней также присутствует очень большое количество (сотни) элементов (прежде всего, драйверы файловых систем и устройств), которые оформлены как внешние модули и которые могут подгружаться ядром по мере необходимости. При этом, все компоненты ядра, в том числе подгружаемые модули, работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром.

1.4. Интерфейсы операционной системы

1.4.1. Понятие интерфейса

Определение 1. **Интерфейс** — взаимодействие между объектами по вертикали, как вышестоящего с нижестоящим, как старшего с младшим.

Определение 2. **Интерфейс** — совокупность аппаратно-программного обеспечения и документации, определяющего взаимодействие между вышестоящими и нижестоящими уровнями в одной системе.

Для внешней среды (пользовательским процессам) ядро операционной системы предоставляет интерфейс виртуальной машины (см. рис. 5).

Пользовательские процессы (системное и прикладное программное обеспечение) пишутся (программируются) без каких-либо знаний о физическом оборудовании, имеющемся на данном компьютере: ядро абстрагирует все особенности оборудования в четком виртуальном интерфейсе — API (Application Programming Interface — интерфейс прикладного программирования). Принципы построения API в разных операционных системах различны: в одних API точно определён и почти стабилен, в других — переменчив.

Другими словами, **API операционной системы — набор команд виртуальной машины**, в терминах которых программисты и сочиняют свои программы для этой операционной системы.

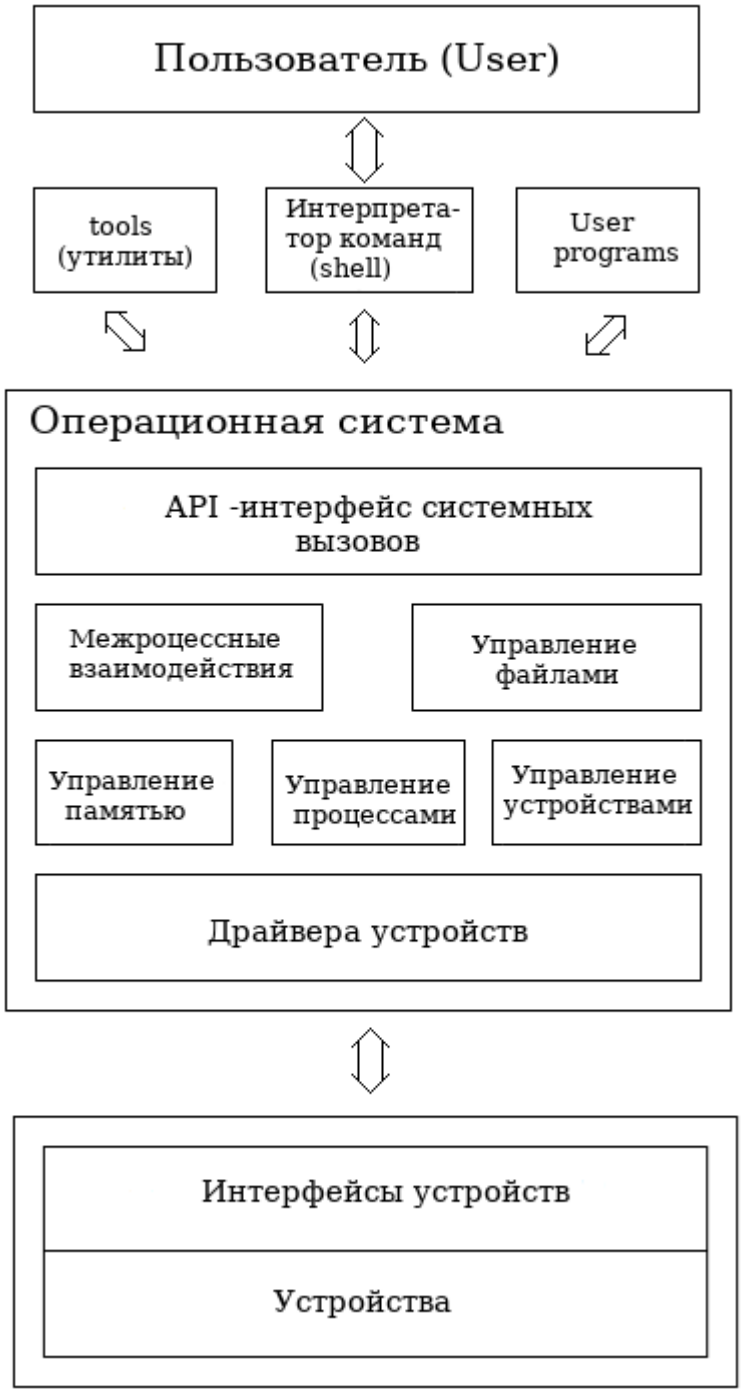


Рис. 5. Интерфейсы операционной системы

1.4.2. Системные вызовы

Интерфейс прикладного программирования API (Application Programming Interface) — это некоторый механизм, который позволяет пользовательским программам обращаться за услугами ядра операционной системы. Этот механизм состоит из системных вызовов (см. рис. 6). Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Физически системный вызов представляет собой имя функции ядра, видимое извне, из-за пределов операционной системы. Операционные системы Unix написаны на языке Си,

а по правилам этого языка, для того, чтобы некоторая функция была видна глобально, она должна быть объявлена как EXTERN — внешняя. Список этих функций, а также других имён, определённых в программе (меток, глобальных переменных), помещается компилятором в файл system.map — карту глобальных имен программы с адресами привязки. В силу того, что операционная система очень большая программа, в файле system.map наличествуют многие тысячи имён.

```
1     mov   edx,len     ; len – длина строки символов
2     mov   ecx,msg     ; msg – адрес строки символов
3     mov   ebx,1       ; дескриптор файла: 1 – это файл stdout
4     mov   eax,4       ; номер системного вызова: 4 – это вызов write
5     int   0x80        ; команда программного прерывания
6
7     mov   eax,1       ; номер системного вызова: 1 – это вызов exit
8     int   0x80        ; команда программного прерывания
9
10    section .data     ; объявление сегмента данных
11    msg db      'Hello, world!',0xa
12    len   equ $ - msg ; длину строки помещаем в len
```

Рис. 6. Системные вызовы: пример программы.

Зная имя функции, что делает функция, и какие параметры обрабатывает функция, можно вызвать функцию, передав её нужные данные, и тем самым, заставить её что-то сделать. В этом смысл системного вызова.

Однако, это же чужая функция, функция другой программы. А вызвать из некоторой программы функцию другой программы невозможно — операционная система таких вольностей не позволяет: срабатывает защита памяти (защита процессов друг от друга — одна из важнейших функций ОС). Но в данном случае, этой другой программой является сама операционная система — программа, которая полностью контролирует оборудование. Следовательно, есть возможность вызвать функцию этой чужой программы (операционной системы) специальным образом: в регистры процессора загружаются определенные параметры (код действия и параметры - ссылки на адреса памяти, где реально располагаются данные, которые нужно обработать) и осуществляют прерывание процессора, после чего управление передается обработчику данного вызова (по коду действия), входящему в ядро операционной системы. Код действия является номером системного вызова — именем системного вызова. Или другими словами, код действия — это порядковый номер системного вызова в таблице системных вызовов.

А дальше происходит следующее: операционная система осуществляет проверку — действительно ли произошедшее является системным вызовом. Дело в том, что «приличные и ответственные» операционные системы стараются контролировать поведение пользовательских программ. Среди структур данных таких операционных систем имеется таблица системных

вызовов — в Unix она называется syscalls.master, в Linux — sys_call_table. В этой таблице перечислены **точно** все функции операционной системы, являющиеся системными вызовами. Тем самым API в Unix и Linux точно определён.

Операционная система проверяет, наличествует ли код действия (номер системного вызова) в таблице системных вызовов. Если да, тогда системный вызов (вызванная функция) выполняется и прикладная программа получает результат. Иначе в прикладную программу возвращается код ошибки «неверный системный вызов». Таким образом, несмотря на то, что все внешние функции операционной системы известны (и даже известно, что они делают — ведь исходники доступны), вызваны могут быть только очень ограниченное их число, только те, что входят в API. Так поступают «надёжные» операционные системы.

Пример 1. В операционной системе Linux-2.6.32.10 было 336 системных вызовов, их количество точно определено таблицей sys_call_table и от версии к версии почти не меняется [3]. Таблица доступна по адресу arh/x86/kernel/syscall_table_32.S.

Пример 2. В операционной системе FreeBSD-8.2 было 544 системных вызова, их количество точно определено таблицей syscalls.master и не меняется по крайней мере с 1994 года. Таблица доступна по адресу /sys/kern/syscalls.master.

Пример 3. В операционной системе Windows более 2000 (двух тысяч) системных вызовов, их количество точно не определено и меняется от версии к версии (в NT их было примерно 0x1100, в 8.2 их стало почти 0x1400 [4]). Более того, существуют «недокументированные» системные вызовы, которые программисты тоже могут использовать, если знают как. Наличие «недокументированных» системных вызовов обусловлено тем, что в Windows отсутствует «входной контроль» - можно вызвать любую функцию ядра.

Пример 4. В операционной системе minix-3 определено 29 системных вызовов и они описаны в файле src/include/minix/com.h, а реализуются они задачей SYSTEM, которая спроектирована так, что может обработать только «правильные» системные вызовы.

1.4.3. Прерывания

> Мама (преподаватель на кафедре программного обеспечения, читает курсы по теории операционных систем, архитектуре компьютера и т. п.) ведёт диалог с бабушкой.
> Бабушка: мне нужно, чтобы ты помогла мне со швейной машинкой, а если ты сейчас займёшься покраской, то мне придётся долго ждать, пока ты освободишься.
> Мама: ну, так я прервусь, на то и существуют внешние прерывания.
>
> bash.org.ru

Прерывание *(interrupt)* - событие, генерируемое внешним (по отношению к процессору) устройством. Посредством аппаратных прерываний *(hardware interrupt)* аппаратура либо

информирует центральный процессор о том, что возникло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода-вывода (например, закончено чтение данных с диска в основную память). Важный тип аппаратных прерываний - прерывания таймера, которые генерируются периодически через фиксированный промежуток времени. Прерывания таймера используются операционной системой при планировании процессов (для «квантования» оперативного времени процессора). Каждый тип аппаратных прерываний имеет собственный номер, однозначно определяющий источник прерывания. Аппаратное прерывание - это *асинхронное* событие, то есть оно возникает вне зависимости от того, какой код (какой процесс) исполняется процессором в данный момент. Обработка аппаратного прерывания не должна учитывать, какой процесс является текущим.

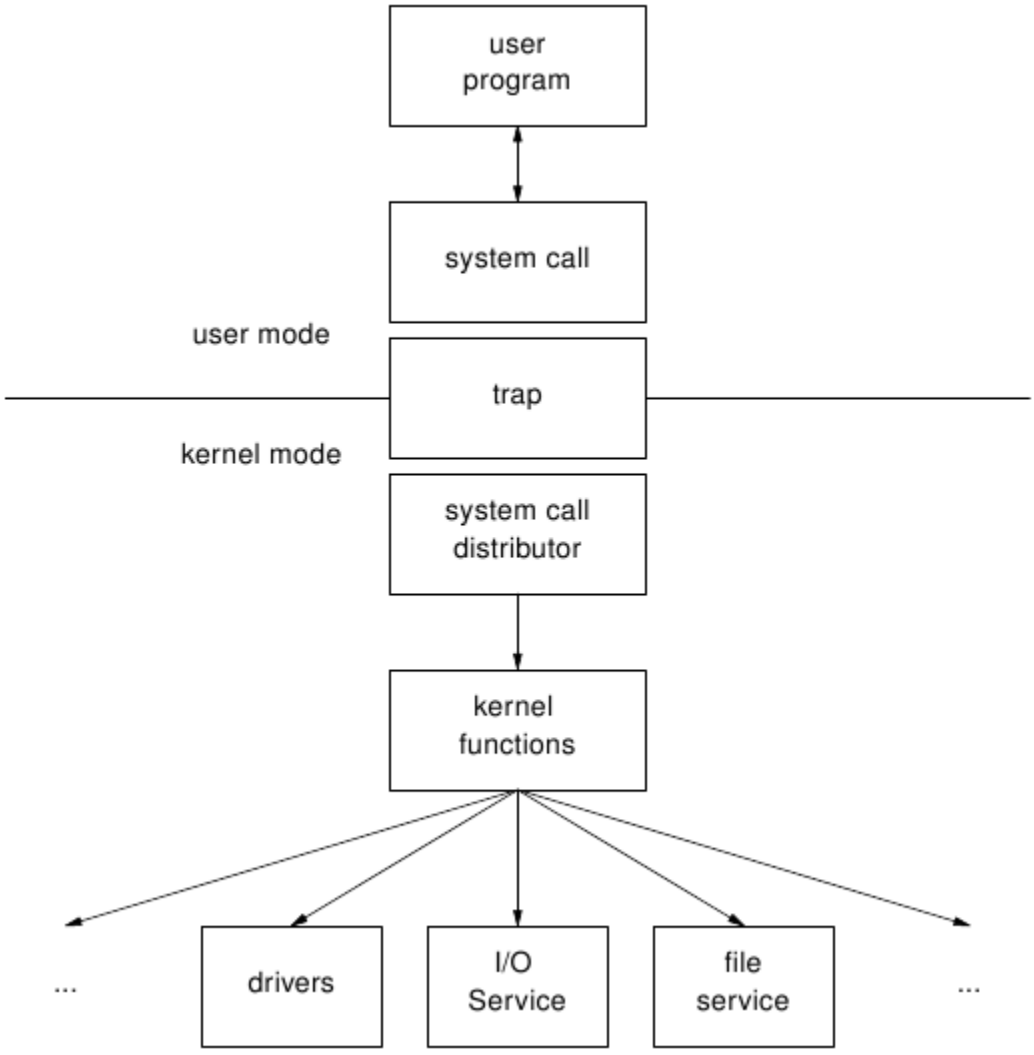


Рис. 7. Программное прерывание — вид сверху.

Программное прерывание (*software interrupt,* см. рис. 7 и рис. 6) — событие, порождаемое программой, некоторым процессом, в том числе, и самой операционной системой. Для

порождения программного прерывания используется специальная команда процессора (int). Если программное прерывание осуществляется некоторым пользовательским процессом, то, почти всегда, это означает системный вызов, то есть, обращение пользовательского процесса к операционной системе с просьбой что-то сделать. Если программное прерывание осуществляется некоторым модулем операционной системы, то это означает обращение этого модуля (вызывающего) к другому модулю (вызываемому) по внутреннему межмодульному интерфейсу операционной системы с просьбой что-то сделать или обработать некоторое событие (см. п. 1.2.2.). Программное прерывание — это *синхронное* событие, оно выполняется в контексте процесса/модуля, породившего прерывание. При этом процесс/модуль приостанавливается на время обработки программного прерывания и по завершении обработки прерывания управление возвращается в процесс/модуль, породивший прерывание.

Прерывания (и аппаратные, и программные) обрабатываются операционной системой, то есть, за пределами источника прерывания (оборудования или процесса, их породившего). Это означает, что в составе операционной системы должны присутствовать специальные модули — *обработчики прерываний*, причём, специализированные для каждого вида прерывания (для каждого номера).

1.4.4. Исключительные ситуации

Исключительная ситуация *(exception)* - событие, возникающее в результате попытки выполнения программой недопустимой команды, доступа к ресурсу при отсутствии достаточных привилегий или обращения к отсутствующей странице памяти. Исключительные ситуации так же, как и программные прерывания, являются синхронными событиями, возникающими в контексте текущей задачи.

Исключительные ситуации можно разделить на *исправимые* и *неисправимые*. К исправимым относятся такие исключительные ситуации, как отсутствие нужной информации в оперативной памяти (нужной страницы памяти). После устранения причины исправимой исключительной ситуации (например, подкачки нужной страницы из swap'а) программа может продолжить выполнение. Возникновение в процессе работы операционной системы исправимых исключительных ситуаций является нормальным явлением. Неисправимые исключительные ситуации обычно возникают в результате ошибок в программах. Например, деление на нуль или попытка доступа в адресное пространство другой программы. Обычно «правильная» операционная система реагирует на такие ситуации завершением программы, вызвавшей неисправимую исключительную ситуацию.

1.4.5. Файлы

Определение. **Файл - часть пространства на носителе информации, имеющая имя.** Как правило, в этой именованной части пространства хранятся некоторые данные. Либо эта именованная часть пространства носителя резервируется для каких-либо целей.

Для хранения файлов на носителях создаются файловые системы.

*Определение***. Файловая система *(file system)* — метод организации хранения файлов на носителе информации, реализованный в некоторой операционной системе.**

Обратите внимание (!): в определении упоминается «операционная система». Почти всегда файловые системы разрабатываются в рамках проекта некоторой операционной системы и являются принадлежностью (частью) этой операционной системы. В тех редких случаях, когда файловые системы создаются «сами по себе», они всё равно разрабатываются для вполне определённых операционных систем. Также существует зависимость между файловыми системами и носителями, на которых они могут использоваться.

Из того, что файловая система — это «метод», следует, что файловая система реализуется некоторым алгоритмом (реализующим «метод»), который должен работать на некоторых структурах данных («алгоритм + структуры данных = программа» (С) Вирт Н.). Структуры данных определяют область определения алгоритма.

Алгоритм реализуется драйвером файловой системы, являющимся частью операционной системы. Отсюда следует «операционнозависимость» файловой системы. А структуры данных файловой системы конечно же хранятся на самих носителях. В этих структурах данных описывается, какой это носитель, что за файлы и где хранятся на данном экземпляре носителя.

Назначение файловой системы:

1) «Расшарить» носитель, то есть, обеспечить доступ к носителю разных программ и пользователей.

2) Скрыть особенности ввода-вывода для данного типа носителя и дать программисту простую абстрактную модель файлов, независимых от устройств. Для чтения, создания, удаления, записи, открытия и закрытия файлов в API операционной системы имеется набор системных вызовов (create, delete, open, close, read, write и другие).

Общеизвестны также такие понятия, связанные с организацией файловой системы, как каталог, текущий каталог, корневой каталог, путь, для манипулирования которыми в операционной системе также имеются системные вызовы.

1.5. Понятие процесса в операционной системе

1.5.1. Многозадачность

Современные операционные системы поддерживают многозадачность по методу «прозрачности» для пользовательских процессов: каждый процесс работает так, как будто он один на компьютере и обладает исключительными правами на использование всей памяти и всех имеющихся устройств. В действительности же ядро операционной системы запускает несколько (десятки и сотни) процессов при старте и возможно ещё очень много по указанию пользователя, и в задачу ядра входит посредничество между процессами и некими аппаратными и программными ресурсами (сервисами), так что каждый из процессов имеет доступ к этим ресурсам.

1.5.2. Определение процесса

Определение краткое. **Процесс** — это запущенная на исполнение программа.

Определение сложное. **Процесс** — это объект, создаваемый операционной системой при запуске некоторой программы (исполняемого файла) и который состоит из двух элементов:

- контекста процесса — адресного пространства, объёмом 4 Гб (на 32-разрядных ЭВМ), в котором операционная система, на основании информации из исполняемого файла, размещает коды программы, описания структур данных, стек,
- блока управления процессом (PCB — Process Control Blok), в котором операционная система хранит (и сохраняет) всю необходимую информацию для управления данным процессом (рис. 8).

Что такое контекст процесса — это почти понятно.

Что такое PCB и зачем он нужен — тоже становится понятным, если вспомнить, что в современных операционных системах «одновременно» выполняются десятки, сотни (и даже тысячи) процессов, что «Планировщик процессов», распределяя процессорное время, то останавливает процессы (естественно, с запоминанием их состояния), то вновь их запускает (восстанавливая их состояние). Чтобы иметь возможность эту работу выполнять, а также много других действий с процессами, «Планировщик процессов» должен иметь возможность где-то запоминать всё, что происходит с процессами. Этим местом, куда «Планировщик процессов» записывает всё о процессах, являются PCB процессов.

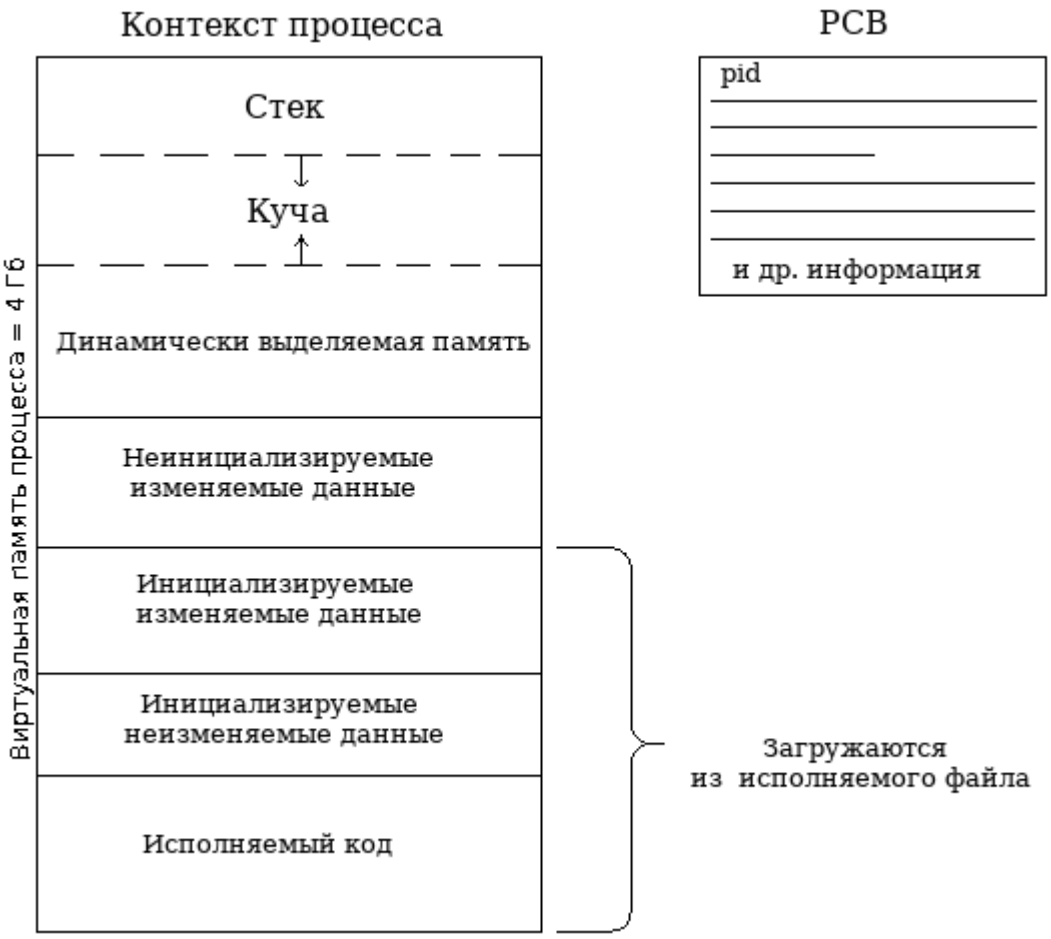


Рис. 8. Структура процесса.

PCB — это достаточно сложная структура, точно она называется task_struct (см. Приложение 1) и в ней хранится (и сохраняется) следующая информация:

- идентификаторы: каждый процесс имеет уникальный идентификатор (pid), а также пользовательский и групповой id; групповой id используется для назначения прав группе процессов;

- state: состояние или статус выполнения процесса (выполнение, чтение, спит, остановлен, зомби);

- информация для планировщика, необходимая для управления процессами:

= приоритет,

= признак real-time, если да, то процесс имеет преимущества перед нормальными процессами,

= counter - количество времени, отводимое процессу на выполнение;

- параметры межпроцессного взаимодействия: Linux поддерживает IPC-механизм Unix SVR4;

- линки: каждый процесс включает линк на его родительский процесс, кроме того, наследует линки (siblings) на другие процессы с тем же родителем, а также линки на всех своих детей;

- times и timers: время создания процесса; PCB может включать в себя таймеры, которые привязаны к системному вызову; когда таймер истекает, процессу посылается сигнал; таймер может быть однократный либо периодический;

- файловая система: указатели на файлы, открытые данным процессом, а также указатели на собственный исполняемый файл и текущий (рабочий) каталог;

- адресное пространство: виртуальное адресное пространство процесса; адрес таблицы страниц;

- данные текущего контекста процесса: регистровая и стековая информация;

- статус выполнения: два состояния - выполняемый либо готовый к исполнению;

- прерывание: уточнение состояния ожидания, в котором находится процесс, ожидающий какого-то события, например, ожидает конца инициированной I/O операции или сигнала от другого процесса;

- uninterruptible: другое блокировочное состояние процесса; например, процесс ждет сигнала от железа и не реагирует более ни на что;

- stopped: остановленный процесс, например, в процессе дебага;

- зомби: процесс убит или завершился, а его task structure до сих пор «болтается» в таблице процессов.

Из всех PCB процессов «Планировщик процессов» организует списковую структуру task_list. Это структура постоянно меняется «Планировщиком процессов» в соответствии с алгоритмом распределения процессорного времени (алгоритмом «Планировщика процессов») и состоянием процессов. Очень часто в литературе эту структуру также называют *очередью задач* к процессору.

Операционной системой для работы некоторых программ может организовываться более одного процесса или наоборот, один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса, нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы и поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле программы), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ими ситуациях (например, при обработке внешних прерываний или исключительных ситуаций).

1.5.3. Жизненный цикл процесса

В операционных системах Unix/Linux все процессы, кроме процесса init, создающегося при

старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве процесса-прародителя всех остальных процессов выступает процесс init с id, равным 0 или 1.

Таким образом, все процессы в Unix/Linux связаны отношениями процесс-родитель - процесс-потомок, образуя **связное** генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-потомка, идентификатор родительского процесса (PPID - Parent Process Identificator) в PCB процесса-потомка изменяет свое значение на значение 1, соответствующее идентификатору процесса *init*, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы усыновляет осиротевшие процессы и тем самым сохраняется связность генеалогического дерева.

Жизненый путь процесса в вычислительной системе начинается с его «рождения» (рис. 9), то есть, операционной системой создаётся контекст процесса и PCB. При рождении процессу присваивается имя — идентификатор процесса PID (*Process Identificator*) — беззнаковое целое в диапазоне от 0 до 65535. PCB включается в task_list, то есть, ставится в очередь к процессору — это новое состояние называется «готовность» к исполнению.

Рис. 9. Жизненный цикл процесса

Когда «Планировщик процессов» выбирает процесс для исполнения, то сначала процесс «исполняется в режиме ядра». В этом состоянии выполняются коды ядра операционной системы в пользовательском контексте (в «пространстве пользователя») - ядро восстанавливает контекст процесса на основе информации из PCB, готовит процесс к выполнению (активизируются

необходимые страницы памяти, восстанавливаются регистры процессора, восстанавливается счётчик команд).

Когда всё готово, управление передаётся процессу (по счётчику команд) и начинают исполняться команды процесса (те коды, что написал программист) - «исполнение в режиме пользователя».

Далее возможны две цепочки событий:

1) **С системным вызовом.** Процесс сам себя прерывает.

В процессе выполнения процесс может сделать системный вызов. В этом случае снова выполняются коды ядра операционной системы в пользовательском контексте (в «пространстве пользователя») - ядро сохраняет контекст процесса в PCB, а процесс переводится в состояние «ожидания» завершения инициированного им системного вызова.

В состоянии «ожидания» процесс может находиться неопределённое время, в зависимости от того, какую операцию он инициировал: если это обмен с диском, то 5-12 миллисекунд, а если это ввод с клавиатуры в пятницу в конце рабочего дня?

Тем не менее, когда инициированная операция завершится, произойдёт прерывание и операционная система определит, какой процесс ожидает это событие. Этот процесс будет переведён снова в состояние «готовность», то есть снова ставится в очередь к процессору.

2) **С истечением кванта времени.** Процесс прерывается насильно.

В процессе выполнения в некоторый момент квант времени, предоставленный процессу, заканчивается и происходит прерывание от таймера. Выполняются коды ядра операционной системы в пользовательском контексте (в «пространстве пользователя») - ядро сохраняет контекст процесса в PCB, а процесс переводится в состояние «готовности», то есть снова ставится в очередь к процессору.

Эти две цепочки событий происходят до тех пор, пока процесс не дойдёт до своей последней команды. Когда это произойдёт, снова выполняются коды ядра операционной системы (системный вызов exit) в пользовательском контексте (в «пространстве пользователя») - ядро сохраняет контекст процесса в PCB, а процесс переводится в состояние «завершения»

Далее процесс-родитель должен получить и обработать информацию о завершении инициированного им дочернего процесса. Если родитель явно не указал, что он не хочет получать информацию о статусе завершения порожденного процесса, то завершившийся процесс не исчезает из системы сразу по завершении, а остается в состоянии ***«завершение»*** либо до окончания исполнения процесса-родителя, либо до того момента, когда родитель соблаговолит получить эту информацию. Процессы, находящиеся в состоянии ***«завершение»***, в операционной системе принято называть процессами-зомби (*zombie, defunct*). Процессы-зомби характеризуются тем, что они выполняться уже не могут (они же завершились), но ресурсы системы по-прежнему занимают.

Помимо указанных на рисунке 9 шести состояний жизненного цикла процесса есть ещё два состояния, связанных со свопингом (возникают при отсутствии нужной страницы памяти в списке активных страниц или при нехватке памяти) и ещё одно отдельное состояние «изменение приоритета» процесса [6].

1.5.4. Методы порождения процессов

После рождения (при котором прежде всего создаётся PCB процесса) для обеспечения работоспособности необходимо во-первых, определить контекст нового процесса, и во-вторых — наделить процесс необходимыми ресурсами: устройствами ввода-вывода, файлами, средствами взаимодействия и т. д.

Существует два подхода к решению этой задачи:

1) После выделения памяти в адресное пространство нового процесса заносится программный код, значения данных, устанавливается программный счетчик. Это делается

- либо полным копированием содержания контекста родителя в контекст потомка, - тем самым создаётся новый самостоятельный процесс (системный вызов fork),

- либо предоставлением в пользование потомка некоторой части родительских ресурсов (родительского контекста) с возможным разделением с процессом-родителем и другими процессами-потомками прав на них, - в этом случае создаётся легковесный процесс или поток (системный вызов pthread).

Информация о выделенных ресурсах заносится в PCB нового процесса. Таким образом, процесс-потомок становится дубликатом процесса-родителя по контексту более или менее самостоятельным (независимым). Естественно, должен существовать способ определения, кто для кого из процессов-двойников является родителем (это определяется по коду возврата из системного вызова).

2) Новый процесс может получить свои ресурсы непосредственно от операционной системы независимо от процесса родителя и владеть ими полностью самостоятельно. Информация о выделенных ресурсах также заносится в PCB нового процесса. В этом случае процесс-потомок загружается новой программой из какого-либо исполняемого файла (системный вызов exec).

Операционные системы Unix/Linux разрешают порождение процесса только первым способом. В этом случае для запуска совсем новой программы необходимо сначала создать копию процесса-родителя (вызов fork), а затем процесс-потомок должен заменить свой контекст с помощью специального системного вызова exec, в параметрах которого указывается имя исполняемого файла, на основе информации из которого операционная система заменит контекст процесса. Или сразу делается системный вызов execle, который включает в себя вызов fork.

Порождение нового процесса первым способом, как дубликата процесса-родителя,

приводит к возможности существования программ, для работы которых организуется более одного процесса — родственных процессов. Это часто используется для распараллеливания обработки данных в рамках одной программы с минимальными накладными расходами. А возможность замены контекста процесса по ходу его работы (т. е. загрузки для исполнения новой программы) приводит к тому, что в рамках одного и того же процесса могут быть последовательно выполнены несколько различных программ.

Операционная система Windows допускает только второе решение. Понятие «родственный процесс» в Windows отсутствует и поэтому выше накладные расходы на создание нового процесса. Это приводит к тому, что в среде Windows более распространено распараллеливание обработки посредством создания потоков в процессе.

1.5.5. Планирование процессов

Долгосрочное планирование (планирование заданий) определяет то, как в системе создаются новые процессы. Оно определяет степень мультипрограммирования, то есть, сколько процессов одновременно находятся в системе. Поскольку создание нового процесса - событие относительно редкое и определяет поведение системы на протяжении достаточно длительного интервала времени, поэтому такое планирование называется долгосрочным.

Среднесрочное планирование определяет работу подсистемы свопинга (*swapping).* В рамках этого планирования решается когда и какой из процессов нужно перекачать на диск (сбросить в swap-раздел) или вернуть обратно, например, в целях повышения производительности системы. Но необходимость сброса процесса на диск (в swap) возникает только тогда, когда начинает не хватать памяти. Если в ЭВМ памяти достаточно, то и среднесрочное планирование не задействуется.

Краткосрочное планирование — это планирование использование процессора. Оно реализуется, например, когда процесс делает системный вызов или при завершении очередного кванта времени и приводит к выбору нового процесса для исполнения.

Эти виды планирования определяют работу подсистемы «Планировщик процессов».

Цели планирования:

1) Справедливость: гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе.

2) Эффективность: постараться занять процессор на все 100% рабочего времени.

3) Сокращение полного времени выполнения процессов.

4) Сокращение времени ожидания в очереди в состоянии «готовность».

5) Сокращение времени отклика системы на события, в том числе, на запросы пользователя.

Реализуется два типа планирования:

1) **Кооперативное (**невытесняющее), при котором процесс занимает столько процессорного времени, сколько ему необходимо. То есть, переключение процессов возникает только при желании самого исполняющегося процесса передать управление (при системном вызове или по окончании работы).

Этот метод планирования относительно просто реализуем и достаточно эффективен, так как позволяет использовать большую часть процессорного времени на работу самих процессов и до минимума сократить затраты на переключение контекста (поэтому иногда используется в системах реального времени).

Однако при таком планировании возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

2) **Вытесняющее** планирование обычно используется в системах разделения времени. В этом режиме планирования процесс может быть приостановлен в любой момент своего исполнения. Операционная система устанавливает специальный таймер для генерации сигнала прерывания по истечении некоторого интервала времени — *кванта*. После прерывания процессор передается в распоряжение следующего процесса.

Временные прерывания помогают гарантировать приемлемые времена отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают “зависание” компьютерной системы из-за зацикливания какой-либо программы.

1.5.6. Алгоритмы планирования

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой ***FCFS*** — First Come, First Served (первым пришел, первым обслужен). Иногда этот алгоритм называется также FIFO - First In, First Out (первым вошел, первым вышел).

Этот алгоритм выбора процесса осуществляет невытесняющее планирование. То есть, процессы, находящиеся в состоянии готовность, организованы в очередь. Когда процесс переходит в состояние готовность, он, а точнее ссылка на его PCB, помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Процесс, получивший в свое распоряжение процессор, занимает его до тех пор, пока не решит сам освободить процессор по какой либо причине.

Алгоритм FCFS имеет много модификаций. Например, но его основе может быть реализован алгоритм «кратчайшая работа — первая» (***SJF*** — *Shortest Job First*), в котором планировщик некоторым способом пытается оценить продолжительность исполнения процессов и

реорганизовать очередь процессов так, чтобы короткие процессы получили доступ к процессору первыми. Цель — быстрее разгрузить систему, минимизировать количество процессов, стоящих в очереди. Сложность состоит в том, как определить, что процесс быстро завершится?

В современных многозадачных системах обычно используется алгоритм гарантированного планирования с динамическими приоритетами с вытеснением. Смысл его в следующем:

1) Все процессы при рождении получают приоритеты.

2) Планировщик ведёт столько очередей, сколько приоритетов определено в системе.

3) Процесс в состоянии «готовность» ставится в очередь своего приоритетного уровня.

4) Планировщик при распределении каждого кванта начинает просматривать очереди снизу и если в низкоприоритетной очереди обнаруживается процесс Z, долгое время не получавший доступа к процессору, то ему повышается приоритет на один уровень. Затем планировщик выполняет свой обычный алгоритм, то есть, начинает просматривать очереди сверху.

5) На очередном планировании, если обнаруживается, что процесс Z получил процессорное время и использовал квант, то его приоритет снова понижается до первоначального.

Существует модификация этого алгоритма - алгоритм многоуровневых очередей с обратной связью. Здесь процесс не постоянно приписан к определенной очереди, а может мигрировать из очереди в очередь, в зависимости от своего поведения. Например, если процесс не использует полностью свой квант времени (например, прерывается по системному вызову), то ему может быть повышен приоритет. Смысл подобной модификации в том, чтобы выделить интерактивные процессы и, тем самым, улучшить отклик системы (цель пятая планирования — см. п. 1.5.5).

1.5.7. Потоки

Поток — это некоторая абстракция внутри понятия «процесс». Потоки процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждый поток имеет свой PCB, в котором запоминаются счетчик команд, содержимое регистров и другая информация (как у полных процессов) и свой собственный стек потока (рис. 10). Поскольку потоки имеют собственные PCB, то ссылки на эти PCB и на PCB их процесса ставятся в очередь Планировщика независимо. Это позволяет нескольким потокам самостоятельно выполняться в рамках одного процесса. Эти потоки выполнения совместно используют ресурсы процесса, но могут работать и самостоятельно.

Многопоточная модель программирования предоставляет разработчикам удобную абстракцию параллельного выполнения. Эта технология стала широко применяться к *одному* процессу, реализуя его *параллельное выполнение* на *многоядерных* процессорах.

Процесс является «самой тяжёлой» единицей планирования ядра. Собственные ресурсы для процесса выделяются операционной системой. Ресурсы включают память, дескрипторы файлов,

дескрипторы устройств, окна. Процессы используют адресное пространство и файлы ресурсов в режиме разделения времени только через явные методы, такие как наследование дескрипторов файлов и сегментов разделяемой памяти. Процессы, как правило, предварительно преобразованы к многозадачному способу выполнения.

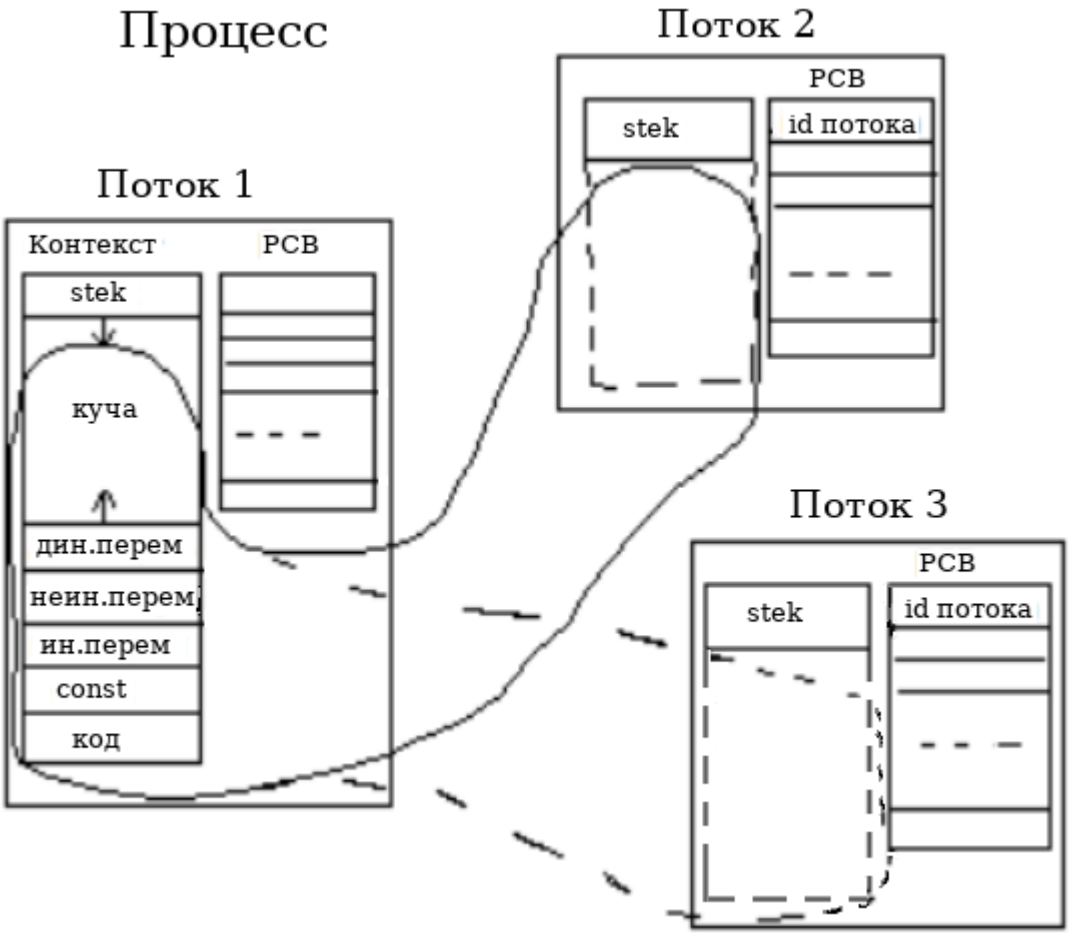


Рис. 10. Потоки: разделение ресурсов внутри процесса.

Потоки выполнения относятся к «лёгким» единицам планирования. Внутри каждого процесса существует по крайней мере один поток выполнения. Если в рамках процесса могут существовать несколько потоков выполнения, то они совместно используют общую память и ресурсы.

То есть, процесс представляется как совокупность взаимодействующих потоков и выделенных ему ресурсов. Процесс, содержащий всего один поток исполнения, идентичен процессу в том смысле, который определён выше. Это - “традиционный процесс”.

Потоки, как и процессы, могут порождать потоки-потомки, правда, только внутри своего процесса, и переходить из состояния в состояние. Состояния потоков аналогичны состояниям традиционных процессов.

Из состояния ***рождение*** процесс приходит содержащим всего один поток исполнения. Другие потоки процесса будут являться потомками этого потока прародителя.

Можно считать, что процесс находится в состоянии ***готовность***, если хотя бы один из его потоков находится в состоянии ***готовность*** и ни один из потоков не находится в состоянии ***исполнение***.

Можно считать, что процесс находится в состоянии ***исполнение***, если один из его потоков находится в состоянии ***исполнение***.

Процесс будет находиться в состоянии ***ожидание***, если все его потоки находятся в состоянии ***ожидание***.

Наконец, процесс находится в состоянии ***завершил исполнение***, если все его потоки находятся в состоянии ***завершили исполнение***.

Пока один поток процесса заблокирован, другой поток того же процесса может выполняться. Потоки разделяют процессор так же, как это делают традиционные процессы, в соответствии с рассмотренными выше алгоритмами планирования.

Поскольку потоки одного процесса разделяют существенно больше ресурсов, чем различные процессы, то операции создания нового потока и переключения контекста между потоками одного процесса занимают существенно меньше времени, чем аналогичные операции для процессов в целом. Следовательно, схема распараллеливания обработки информации с помощью потоков внутри одного процесса вполне эффективна.

Различают операционные системы, поддерживающие потоки на уровне ядра (Linux, FreeBSD) и на уровне библиотек (Windows).

Все выше сказанное в данном пункте справедливо для операционных систем, поддерживающих потоки на уровне ядра (Linux). В них планирование использования процессора происходит в терминах потоков, а управление памятью и другими системными ресурсами остается в терминах процессов.

В операционных системах, поддерживающих потоки на уровне библиотек языка (Windows), и планирование процессора, и управление системными ресурсами осуществляется в терминах процессов. Распределение использования процессора по потокам в рамках выделенного процессу временного интервала осуществляется средствами библиотеки. В таких системах блокирование одного потока приводит к блокированию всего процесса, завершение потока означает завершение всего процесса и всех его потоков, ибо ядро операционной системы ничего не знает о существовании потоков в прикладных программах. По сути дела, в таких вычислительных системах просто имитируется наличие потоков исполнения.

1.6. Управление памятью

1.6.1. Введение в управление памятью

Чтобы программы могли выполняться, они, вместе с данными, которые они обрабатывают, должны (по крайней мере, частично) находиться в физической (оперативной) памяти. Та часть операционной системы, которая управляет оперативной памятью, называется «**Менеджер памяти**».

Основные идеи, на которых основываются схемы управления памятью:

1) **Сегментация** — память делится на отдельные участки — сегменты, которые операционная система отображает а адресное пространство нескольких процессов с целью исключения дублирования. В сегменты помещаются общие фрагменты кода (чаще всего, библиотеки) или данные разных типов (код программы, данные, стек и т. д.). Операционная система контролирует характер работы с сегментами, назначая им атрибуты, например, права доступа или типы операций, разрешенные с данными, хранящимися в сегменте. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента.

2) **Разделение памяти** *на физическую и логическую*. Адреса, к которым обращается процесс, как правило, не те, что реально существуют в оперативной памяти. Адрес, сгенерированный программой, обычно называют *логическим (виртуальным)* адресом, тогда как адрес, который видит устройство памяти (то есть нечто, загруженное в адресный регистр) обычно называется *физическим* адресом. Задача операционной системы, в какой-то момент времени осуществить связывание (или отображение) логического адресного пространства программы с физическими адресами, реально существующими в системе.

3) **Локальность** — смысл её в том, что обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Понимание данной особенности позволяет организовать *иерархию памяти*, используя быструю дорогостоящую память для хранения минимума необходимой информации, размещая оставшуюся часть данных на устройствах с более медленным доступом и подкачивая их в быструю память по мере необходимости. Типичный пример иерархии: регистры процессора <--> кэш процессора <--> оперативная память <--> внешняя память на магнитных дисках (*вторичная память*).

4) **Swapping** — использование вторичной памяти (на магнитных дисках) в качестве расширения оперативной памяти даёт дополнительные преимущества вычислительной системе: поскольку оперативной памяти всегда не хватает, то появляется возможность неиспользуемые части программ сбросить (swap out) в специальный раздел диска (раздел swap, в Windows - просто файл), подкачивая их по мере необходимости. Тем самым, оперативная память освобождается и в ней могут быть размещены дополнительные программы. К тому же снимаются ограничения на размер программ.

К функциям операционной системы по управлению памятью («менеджера памяти») относятся:

- отображение адресов программы на конкретную область физической памяти,
- распределение памяти между процессами и защита адресных пространств процессов,
- выгрузка процессов на диск, в swap-раздел, и подкачка обратно по необходимости,
- учет свободной и занятой памяти.

Управление памятью в вычислительной системе реализуется на двух уровнях:

- на аппаратном уровне работает аппаратный менеджер памяти — он непосредственно управляет страничками физической памяти; на современных интеловских микропросессорах микросхема менеджера памяти интегрирована в один блок с микропроцессором (в «камень»);
- на программном уровне работает подсистема операционной системы «Менеджер памяти» — она управляет логической (виртуальной) памятью процессов (см. п. 1.2.2 и рис. 2).

1.6.2. Связывание адресов

Связывание (или отображение) логического адресного пространства программы с физическими адресами, реально существующими в системе, обычно понимается как преобразование адресных пространств (рис. 11).

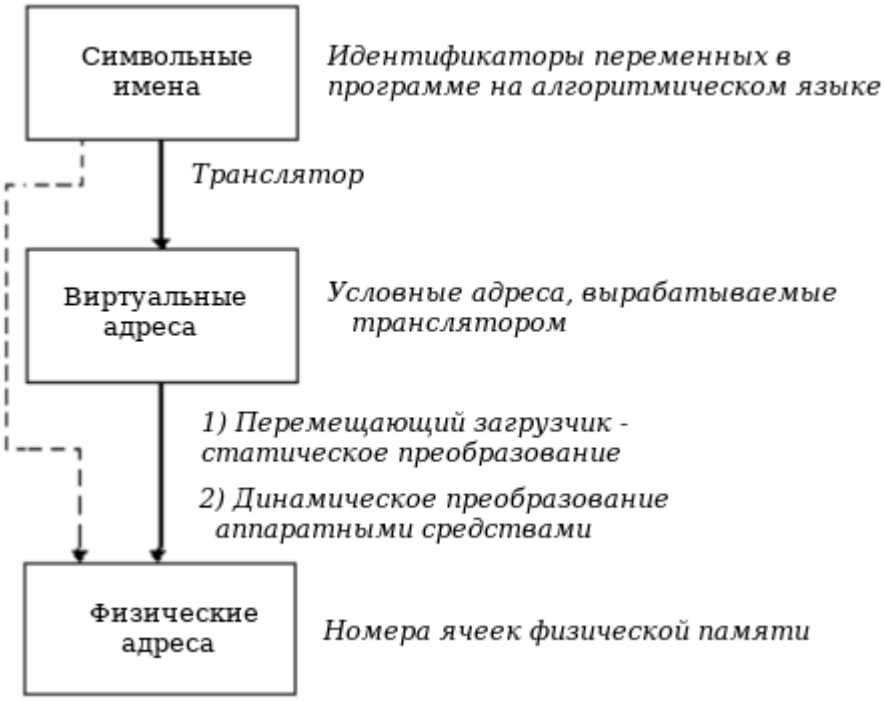


Рис. 11. Преобразование адресных пространств.

Пользовательская программа работает с логическими адресами, которые являются результатом трансляции символьных имен программы (которые «сочинил» программист).

Логические адреса генерятся на этапе создания загрузочного модуля (линковки программы) компилятором и образуют *логическое (виртуальное) адресное пространство*, которому потом, на этапе выполнения, должно соответствовать *физическое адресное пространство*. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 2**32 = 4 Гб) и на ПЭВМ часто превышает размер физического адресного пространства.

Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Например, это может быть сделано:

1) На этапе компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда генерируются абсолютные адреса. Если стартовый адрес программы меняется, необходимо перекомпилировать код. Пример: программы *.com операционной системы MS-DOS, в которых уже на этапе компиляции порождаются физические адреса имён (меток, имён функций, структур данных и т. д.).

2) На этапе загрузки (Load time). Если на стадии компиляции не известно, где процесс будет размещен в памяти, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки и реализуется перемещающим загрузчиком. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.

3) На этапе выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одного сегмента памяти в другой, связывание откладывается до времени выполнения. Здесь используется специализированное оборудование, например регистры перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом.

1.6.3. Виртуальная память

Виртуальная память - это реально не существующая условная память, и она появилась как решение задачи размещения в памяти программ, размер которых превышает размер доступной физической памяти, тем более, если таких программ много. Понятие виртуальной памяти базируется на принципе локальности: для реальных программ обычно нет необходимости размещать их в физической памяти целиком всё время.

Определение. **Виртуальная память** - это совокупность программно-аппаратных средств, позволяющих пользователям писать программы, размер которых превосходит имеющуюся оперативную память; для этого виртуальная память решает следующие задачи:

- размещает данные в запоминающих устройствах разного типа, например, часть программы в оперативной памяти, а часть на диске;
- перемещает по мере необходимости данные между запоминающими устройствами разного

типа, например, подгружает нужную часть программы с диска в оперативную память;

- преобразует виртуальные адреса в физические.

Все эти действия выполняются *автоматически*, без участия программиста, то есть механизм виртуальной памяти является прозрачным по отношению к пользователю.

Кроме того, использование виртуальной памяти позволяет решать ещё одну важную задачу: обеспечение контроля доступа к отдельным сегментам памяти и в частности *защиту* пользовательских программ друг от друга и защиту ОС от пользовательских программ.

Реализация виртуальной памяти получила широкое развитие после появления в процессорах соответствующей аппаратной поддержки. Эта поддержка состоит в том, что адрес памяти, вырабатываемый командой, интерпретируется аппаратурой не как реальный адрес некоторого элемента оперативной памяти, а как некоторая структура, где адрес является лишь одним из компонентов наряду с атрибутами, характеризующими способ обращения по данному адресу и принадлежность этого адреса какому-либо объекту. Механизм преобразования виртуальных адресов в физические должен предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в текущий момент находятся в физической памяти и где именно размещаются. Если такое отображение осуществлять побайтно, то информация об отображении была бы велика, и для ее хранения потребовалось бы больше реальной памяти, чем для процессов. Поэтому обычно отображаемая информация группируется в блоки-страницы и программа занимает целое количество страниц памяти.

1.6.4. Организация памяти

Страничная организация. Виртуальные адреса делятся на страницы (page), которые в физической памяти образуют страничные кадры (page frames). В целом, система поддержки страничной виртуальной памяти называется *пейджингом* (paging).

Передача информации между памятью и диском всегда осуществляется целыми страницами. Страницы, в отличие от сегментов, имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. В самой распространённой сейчас интеловской архитектуре размер страницы 4 кб.

Виртуальный адрес в страничной системе - это упорядоченная пара (p,d), где p - номер страницы в виртуальной памяти, а d - смещение в рамках страницы p, где размещается адресуемый элемент.

Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившая страница может быть размещена в любой свободный страничный

кадр. Система отображения виртуальных адресов в физические сводится к системе отображения виртуальных страниц в физические и представляет собой *таблицу страниц, причём, эти таблицы — отдельные для каждого процесса и ссылки на них хранятся в PCB процессов (рис. 12).*

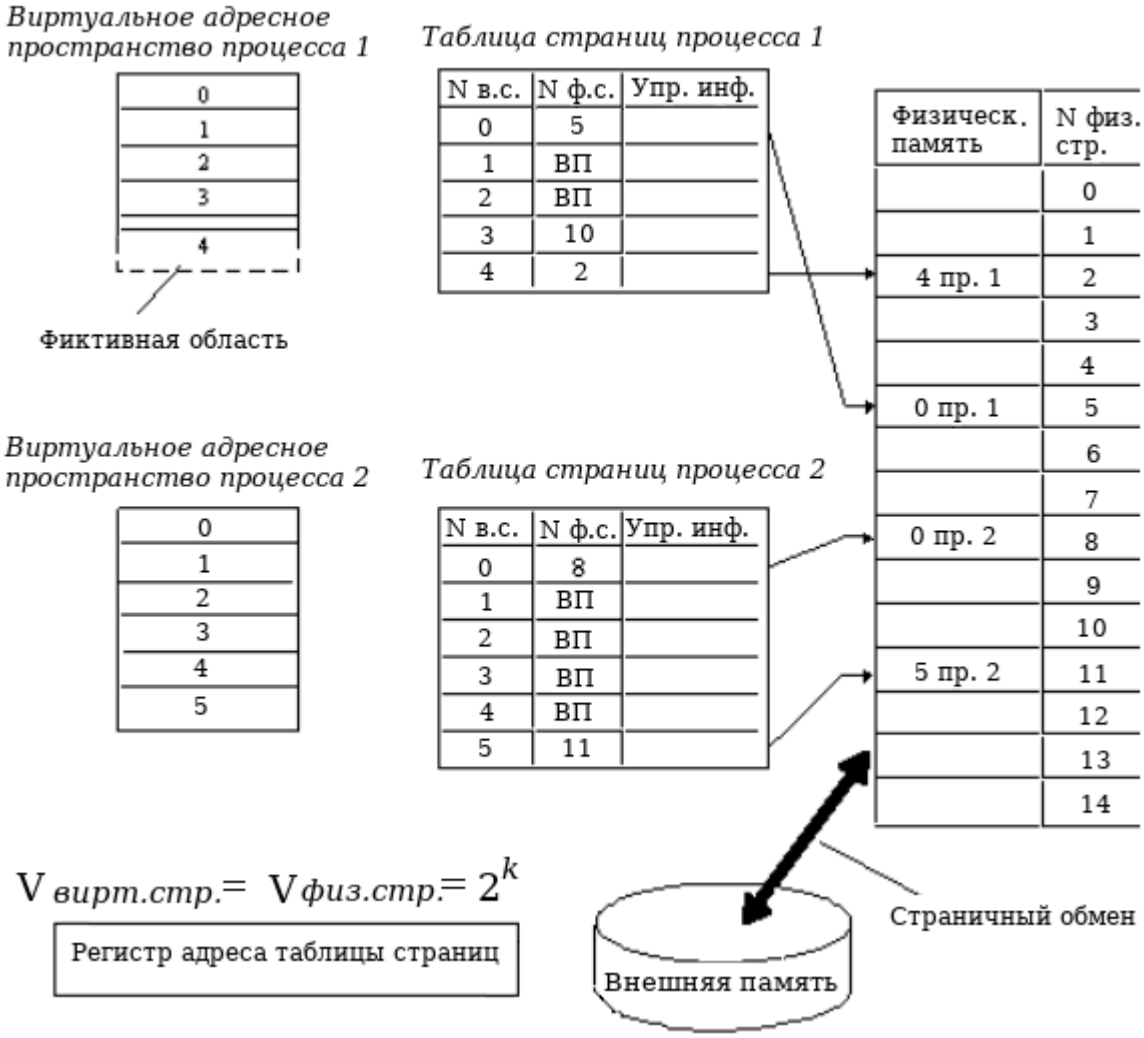


Рис. 12. Отображение страниц памяти процессов

на страницы физической памяти

При создании процесса часть его виртуальных страниц помещается в оперативную память, а остальные - на диск. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. В таблице страниц устанавливается соответствие между номерами виртуальных и физических страниц для страниц, загруженных в оперативную память, или делается отметка о том, что виртуальная страница выгружена на диск. Кроме того, в строках таблицы страниц для каждой страницы содержится управляющая информация (атрибуты страницы, примерно 40 байт), такая как признак модификации страницы, признак невыгружаемости (выгрузка некоторых страниц может быть запрещена), признак обращения к странице (используется для подсчета числа обращений за определенный период времени) и другие данные, формируемые и используемые механизмом виртуальной памяти. С помощью атрибутов отмечается, какому процессу принадлежит страница, режим доступа к странице (только чтение,

выполнение, изменение), признак присутствия страницы (связанности с физической страницей), признак модификации (если на неё писали), признак ссылки (когда на неё ссылались, востребована ли она) и т. д.

При активизации очередного процесса из PCB в специальный регистр процессора загружается адрес таблицы страниц данного процесса.

При каждом обращении к памяти происходит чтение из таблицы страниц информации о виртуальной странице, к которой произошло обращение. Если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический (см. рис. 13).

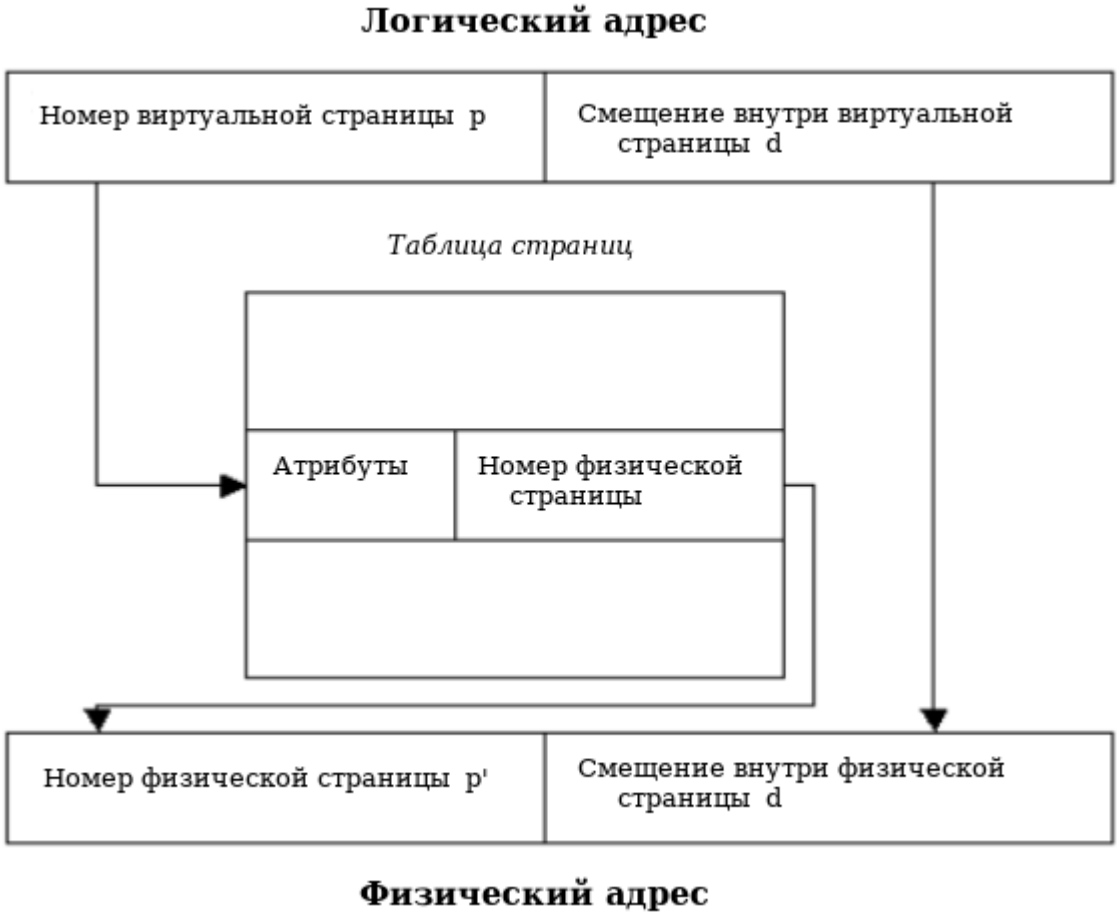


Рис. 13. Связь логического и физического адресов при страничной организации памяти

Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит *страничное прерывание (page fault)*. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди готовых. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то решается вопрос, какую страницу следует выгрузить из оперативной памяти.

При использовании схемы пэйджинга, отсутствует внешняя фрагментация. Однако

наличествует *внутренняя фрагментация*: адресное пространство процесса занимает целое число сегментов, сегмент равен целому числу страниц и в среднем половина страницы на сегмент пропадает.

Важный аспект - различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя его память - единое непрерывное пространство, содержащее, только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. При этом, процессу пользователя недоступна чужая память. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

А для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающую его состояние.

Отображение должно происходить корректно даже в сложных случаях. ОС поддерживает одну или несколько (если виртуальное пространство процесса разбито на нескольких сегментов памяти — как правило, так и бывает) таблиц страниц для каждого процесса, для ссылки, на которую, обычно используется специальный регистр. При переключении процессов диспетчер должен найти его таблицу страниц, указатель на которую, таким образом, входит в контекст процесса (сохраняется в PCB).

Сегментная организация. С точки зрения ОС сегменты, на которые разбита виртуальная память процесса, являются логическими сущностями и их главное назначение - хранение и защита однородной информации (кода, данных и т.д.).

С точки зрения пользователя процесс представляется обычно не как линейный массив байтов, а как набор сегментов переменного размера (код, данные, стек). Сегментация - схема управления памятью, поддерживающая этот взгляд пользователя. Сегменты содержат функции, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа за одним исключением: программисты, пишущие на языках низкого уровня должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере).

Логическое адресное пространство - набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). Пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением. (В отличие от схемы пэйджинга, где пользователь задает только один адрес, который разбивается hardware на номер страницы и смещение, прозрачным для программиста образом). Логический адрес - упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента (рис. 14).

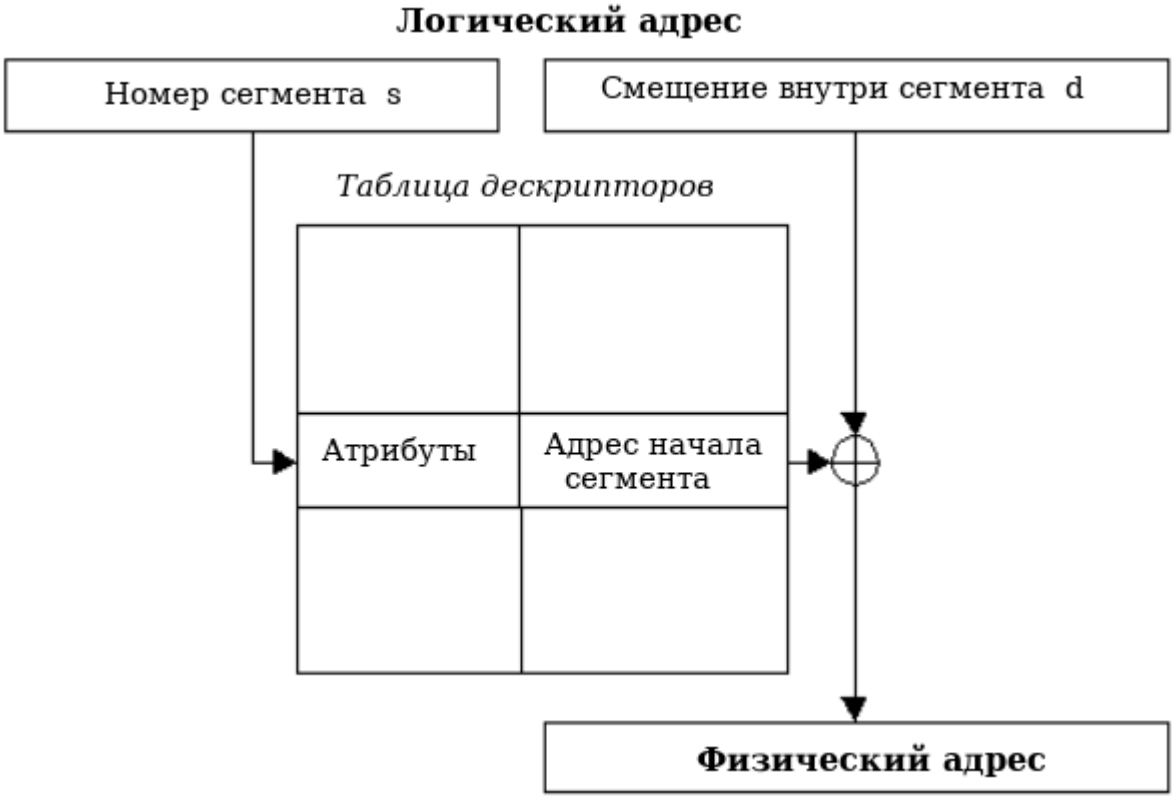


Рис. 14. Преобразование логического адреса
при сегментной организации памяти

Каждый сегмент - линейная последовательность адресов от 0 до максимума. Различные сегменты могут иметь различные длины, которые могут меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента (если виртуальный сегмент содержится в основной памяти) содержится длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает прерывание.

В системах, где сегменты поддерживаются аппаратно (intel), эти параметры обычно хранятся в таблице *дескрипторов* сегментов, а программа обращается к этим дескрипторам по номерам-*селекторам*. При этом в контекст каждого процесса входит набор *сегментных регистров,* содержащих селекторы текущих сегментов кода, стека, данных и др. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

Сегментно-страничная организация. Хранение в памяти сегментов большого размера может оказаться неудобным. Возникает идея их пейджинга. При *сегментно-страничной* организации виртуальной памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае виртуальный адрес состоит из трех полей: номера сегмента виртуальной памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения - таблица сегментов, связывающая номер

сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента (рис. 15).

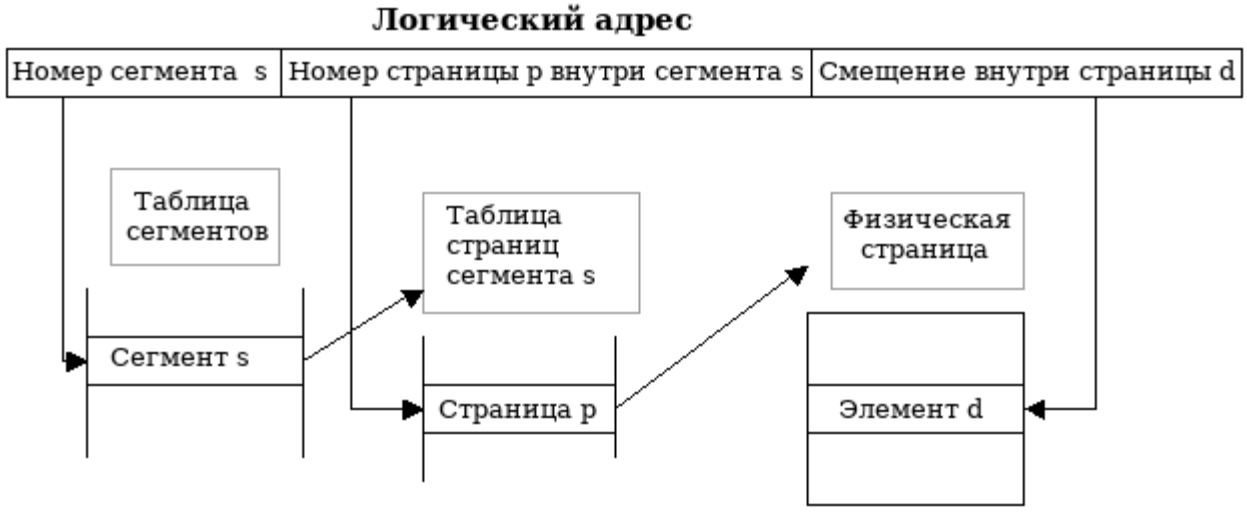


Рис. 15. Формирование физического адреса
при сегментно-страничной организации памяти

Сегментно-страничная организация виртуальной памяти позволяет совместно использовать одни и те же сегменты данных и программного кода в виртуальной памяти разных задач (для каждой виртуальной памяти существует отдельная таблица сегментов, но для совместно используемых сегментов поддерживаются общие таблицы страниц).

1.6.5. Свопинг

Загрузка процессора зависит от числа одновременно выполняемых процессов и доли времени, проводимого этими процессами в состоянии ожидания ввода-вывода (рис. 16).

Из рисунка видно, что для загрузки процессора на 90% достаточно всего трех вычислительных задач. Однако для того, чтобы обеспечить такую же загрузку интерактивными задачами, выполняющими интенсивный ввод-вывод, потребуются десятки таких задач. Необходимым условием для выполнения задачи является загрузка ее в оперативную память, объем которой ограничен. Для реализации условия используется метод организации вычислительного процесса, называемый свопингом. В соответствии с ним некоторые процессы (обычно находящиеся в состоянии ожидания) временно выгружаются на диск. Планировщик операционной системы не исключает их из своего рассмотрения, и при наступлении условий активизации некоторого процесса, находящегося в области свопинга на диске, этот процесс перемещается в оперативную память. Если свободного места в оперативной памяти не хватает, то выгружается другой процесс.

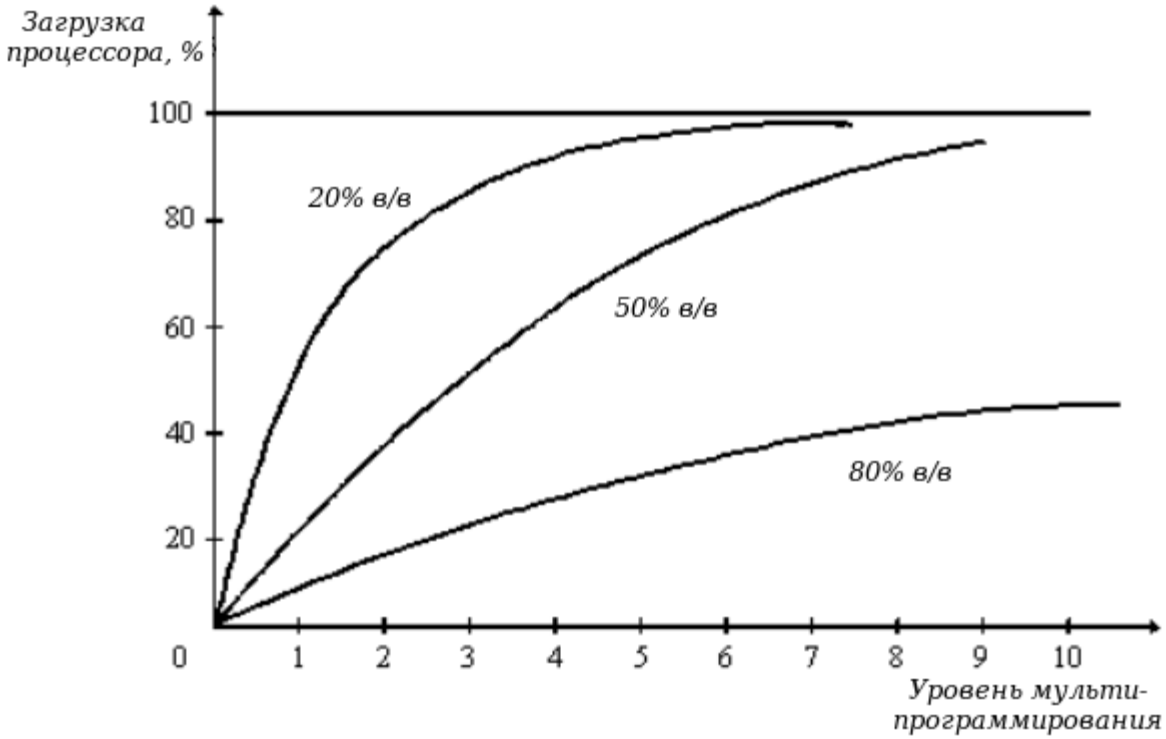


Рис. 16. Зависимость загрузки процессора
от числа задач и интенсивности ввода-вывода

При свопинге, в отличие от рассмотренных ранее методов реализации виртуальной памяти, процесс перемещается между памятью и диском целиком, то есть в течение некоторого времени процесс может полностью отсутствовать в оперативной памяти. Существуют различные алгоритмы выбора процессов на загрузку и выгрузку, а также различные способы выделения оперативной и дисковой памяти загружаемому процессу.

1.6.6. Ассоциативная память и иерархия памяти

Даже для 32-разрядных процессоров с доступным объёмом адресуемого пространства 4 Гб и при размере страницы 4 кб, размер таблицы страниц равен одному миллиону строк, а каждая строка состоит из нескольких байт адреса и атрибутов страницы (обычно 40 байт). Причём такую виртуальную память имеет каждый процесс в системе, а их могут быть сотни и тысячи. То есть, процессору при каждом переключении контекста и даже при каждом обращении в память (что происходит в большинстве машинных команд) необходимо обрабатывать сотни таблиц в млн строк — совсем нетривиальная задача.

Для решения этой задачи, используя свойство локальности, создаются многоуровневые таблицы страниц. При этом используется особенность распределения виртуальной памяти

процессов. Предположим, мы написали достаточно большую программу, которой нужно ~ 12 Мб памяти: 2 Мб в нижней памяти для сегмента кода, 200 байт в нижней памяти для сегмента констант, 8 Мб в нижней памяти для переменных и 2 Мб в верхней памяти для стека. Между верхом данных и дном стека - «куча» байт гигантского неиспользуемого пространства размером (4000-12) Мб, которое, естественно, не нужно адресовать. А для управления страницами в этом случае нужны лишь 4 сравнительно небольших страницы нижнего уровня и одна страница верхнего уровня — ссылочная.

Однако и в этом примере поиск нужной страницы в многоуровневой таблице страниц, требующий несколько обращений к основной памяти, занимает много времени. Чтобы ускорить работу, опять же учитывая свойство локальности (в каждый момент времени только небольшая часть страниц используется), компьютер снабжается аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц — небольшую сверхбыструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство, обычно реализуемое на основе многовходовой памяти, называется ассоциативная память, или ассоциативные регистры, или TLB (translation lookaside buffer — преобразующий «подглядыватель»). Объём TLB обычно небольшой и вмещает от 8 до 2048 строк. Эта память называется *ассоциативной*, потому что благодаря многовходовости происходит одновременное сравнение номера виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. Поэтому эта память является дорогостоящей. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра.

Вполне приемлемая производительность современных операционных систем доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у кода и данных объективных свойств: пространственной и временной локальности.

Однако, при переключении процессов нужно, чтобы новый процесс не видел в ассоциативной памяти информацию, относящуюся к предыдущему процессу, то есть, очищать ее. Это означает, что использование ассоциативной памяти увеличивает время переключения контекстов.

В целом, память ЭВМ представляет собой иерархию запоминающих устройств (внутренние регистры процессора, различные типы сверхоперативной и оперативной памяти, диски, ленты), отличающихся средним временем доступа и стоимостью хранения данных в расчете на один бит (рис.17).

Переход от одного вида памяти к другому достаточно резкий в силу различия физических свойств разных видов памяти. Этот переход несколько смягчается (сглаживается) на основе принципа локальности посредством использования кэш-памяти.

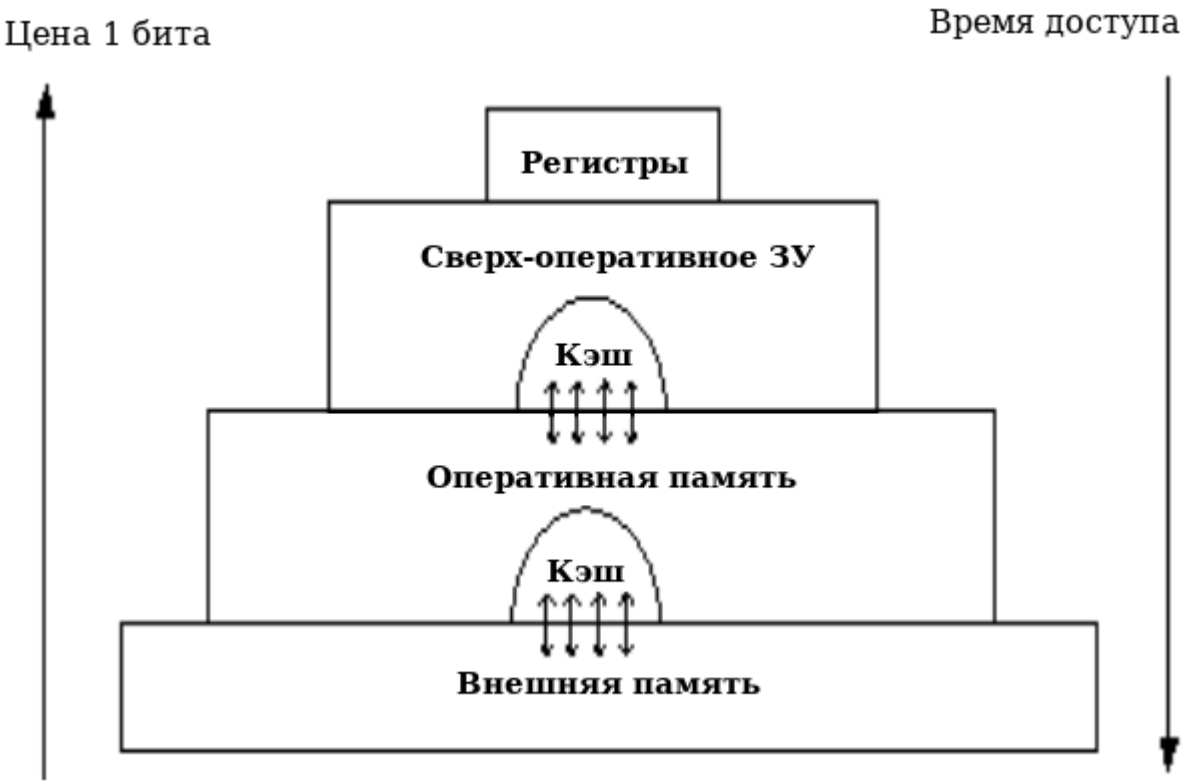
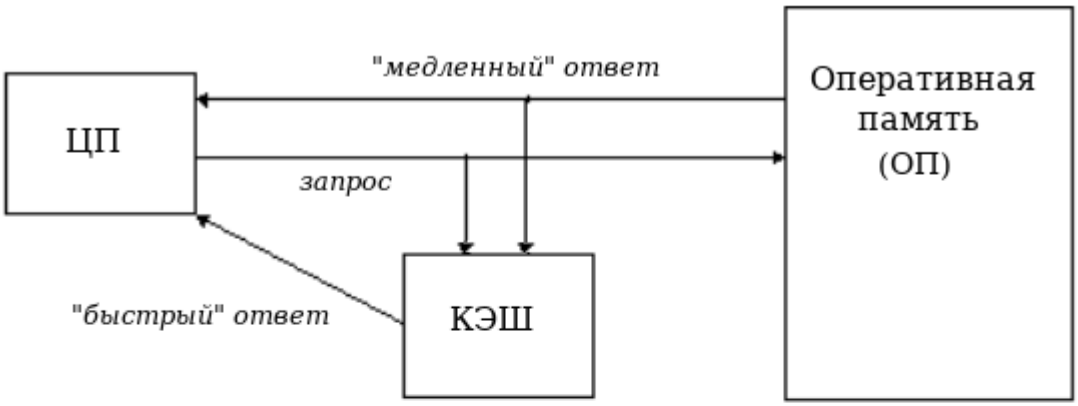


Рис. 17. Иерархия видов памяти.

Кэш-память - это способ организации совместного функционирования двух типов памяти, отличающихся временем доступа и стоимостью хранения данных, который позволяет уменьшить среднее время доступа к данным за счет динамического копирования в «быстрое» ЗУ наиболее часто используемой информации из «медленного» ЗУ.

Кэш-памятью часто называют не только способ организации работы двух типов памяти, но и одно из устройств - «быстрое» ЗУ. Оно стоит дороже и потому, как правило, имеет сравнительно небольшой объем. Важно, что механизм кэш-памяти является прозрачным для пользователя, который не должен сообщать никакой информации об интенсивности использования данных и не должен никак участвовать в перемещении данных из памяти одного типа в память другого типа - все это делается автоматически системными средствами.

Рассмотрим частный случай использования кэш-памяти для уменьшения среднего времени доступа к данным, хранящимся в оперативной памяти. Для этого между процессором и оперативной памятью помещается сверхбыстрая кэш-память (рис.18). В качестве таковой может быть использована, например, ассоциативная память. Содержимое кэш-памяти представляет собой совокупность записей обо всех загруженных в нее элементах данных. Каждая запись об элементе данных включает в себя адрес, который этот элемент данных имеет в оперативной памяти, и управляющую информацию: признак модификации и признак обращения к данным за некоторый последний период времени.



Структура кэш-памяти

Адрес данных в ОП	Данные	Управл. информация	
		бит модиф.	бит обращ.

Рис.18. Кэш-память

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

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

На практике в кэш-память считывается не один элемент данных, к которому произошло обращение, а целый блок данных (в соответствии с принципом локальности), это увеличивает

вероятность так называемого "попадания в кэш", то есть нахождения нужных данных в кэш-памяти.

Аналогичным образом поступают и для других пар запоминающих устройств, например, для оперативной памяти и внешней памяти. В этом случае уменьшается среднее время доступа к данным, расположенным на диске, и роль кэш-памяти выполняет буфер в оперативной памяти.

1.6.7. Управление виртуальной памятью

Общий объём виртуальной памяти процессов многократно превышает объём наличествующей в системе физической памяти. Для того, чтобы процессы могли работать в таких условиях, в строке таблицы страниц устанавливается специальный флаг (означающий отсутствие страницы), наличие которого заставляет аппаратуру, вместо нормального отображения виртуального адреса в физический, прервать выполнение команды и передать управление соответствующему компоненту операционной системы.

Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, т.е. "требует" доступа к данным или программному коду, операционная система удовлетворяет это требование путем выделения страницы основной памяти, перемещения в нее копии страницы, находящейся во внешней памяти, и соответствующей модификации строки таблицы страниц. Это частный случай исключительной ситуации (exception) при работе с памятью, так называемое страничное нарушение (page fault). Другие исключительные ситуации происходят, например, в случаях: при попытке записи в страницу с атрибутом "только чтение" или при попытке чтения или записи страницы с атрибутом "только выполнение" и т. д. При этом, вызывается обработчик (handler) страничного нарушения, являющийся частью операционной системы, которому передается причина возникновения исключительной ситуации и соответствующий виртуальный адрес.

Среди этих ошибок обращения к страницам наиболее длительно обрабатывается ситуация обращения к отсутствующей странице. Время эффективного доступа к отсутствующей странице складывается из:

- времени обслуживания исключительной ситуации — времени работы обработчика прерывания page fault;

- времени чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо предварительно вытолкнуть одну из страниц из основной памяти во вторичную, то есть осуществить замещение страницы);

- времени рестарта процесса, вызвавшего данный page fault.

Первое и третье времена могут быть уменьшены за счет тщательного кодирования нескольких сотен инструкций и составляют, обычно, небольшое время. Время подкачки страницы

с диска будет вероятно близким к нескольким десяткам миллисекунд и определяется средним временем доступа к винчестеру, то есть, по меньшей мере, на три порядка больше. Таким образом, снижение частоты page fault'ов является одной из важнейших задач системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

1.6.8. Стратегии управления страницами

Обычно используются три стратегии: выборка, размещение, замещение.

Стратегия выборки (fetch policy) - в какой момент следует переписать страницу из вторичной памяти в первичную. Выборка бывает по запросу и с упреждением. Алгоритм выборки вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой в данный момент находится на диске (в своп файле или отображенном файле), и потому является ключевым алгоритмом свопинга. Он обычно заключается в загрузке страницы с диска в свободную физическую страницу и отображении этой физической страницы в то место, куда было произведено обращение, вызвавшее исключительную ситуацию.

Существует модификация алгоритма выборки, которая применяет еще и опережающее чтение (с упреждением), то есть, кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (так называемый кластер). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе с большими объемами данных или кода, кроме того, оптимизируется и работа с диском, поскольку появляется возможность загрузки нескольких страниц за одно обращение к диску. Кстати, аналогичным образом оперативная память отображается в кэш-память процессора.

Стратегия размещения (placement policy) — помогает определить, в какое место первичной памяти поместить поступающую страницу. В системах со страничной организацией в любой свободный страничный кадр (в системах с сегментной организацией - нужна стратегия, аналогичная стратегии с переменными разделами).

Стратегия выборки и стратегия размещения используются, если в системе достаточно свободной памяти и нет необходимости задействовать swap.

Стратегия замещения (replacement policy) — определяет, какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место. Разумная стратегия замещения позволяет оптимизировать хранение в памяти самой необходимой информации и тем самым снизить частоту страничных нарушений. Эта стратегия начинает использоваться, если оперативной памяти совершенно недостаточно для размещения страниц всех выполняющихся процессов и операционная система вынуждена задействовать swap.

1.6.9. Алгоритмы замещения страниц

Эти алгоритмы обязательно используются в ситуациях, когда размер виртуальной памяти существенно превосходит размер оперативной памяти и при выделении страницы оперативной памяти с большой вероятностью не удастся найти свободную (не приписанную к виртуальной памяти какого-либо процесса) страницу.

В этом случае операционная система должна сначала в соответствии с заложенными в нее критериями найти некоторую занятую страницу оперативной памяти, переместить в случае надобности ее содержимое во внешнюю память (в swap), должным образом модифицировать соответствующую запись соответствующей таблицы страниц, затем переместить нужную страницу из swap'а в освободившееся место оперативной памяти, опять скорректировать запись соответствующей страницы таблиц и после этого продолжить процесс удовлетворения доступа к странице.

При этом при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения оптимизируется за счет использования бита модификации (один из атрибутов страницы). Бит модификации устанавливается операционной системой, если хотя бы один байт записан на страницу. При выборе кандидата на замещение, проверяется бит модификации. Если бит не установлен, нет необходимости переписывать данную страницу на диск, она уже там. Эта техника также применяется к read-only страницам, они никогда не модифицируются. Эта схема уменьшает время обработки page fault'а.

Все алгоритмы замещения делятся на локальные и глобальные [7,10,15]. **Локальные** алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов.

Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют несколько недостатков. Прежде всего, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе использует большое количество памяти, то все остальные процессы будут в результате ощущать сильное замедление из-за недостатка памяти. Кроме того, некорректно работающее приложение может подорвать работу всей системы (если, конечно, администратор не задействовал login.conf и не определил ограничение на размер памяти, выделяемой процессу), пытаясь захватить все больше памяти. Поэтому в многозадачной системе предпочтительно использовать более сложные, но эффективные локальные алгоритмы. Такой подход требует, чтобы система хранила список физических страниц каждого процесса. Этот список страниц иногда

называют ***рабочим множеством*** процесса.

Рабочее множество процесса должно включать достаточное количество страниц. «Достаточность» - сугубо индивидуальное свойство процесса и определяет некоторое число активно используемых страниц, без которого процесс часто генерирует page fault'ы. Эта высокая частота страничных нарушений называется *трешинг (*trashing - пробуксовка*)*. Процесс находится в состоянии трешинга, если он больше времени занимается подкачкой страниц, нежели выполнением. Критическая ситуация такого рода возникает вне зависимости от конкретных алгоритмов замещения и определяется именно свойствами процесса (точнее, его алгоритма, а если ещё точнее, то качеством программирования). Кстати, пока в вычислительных системах не было поддержки виртуальной памяти, то и проблем такого рода не возникало, в частности, вполне допустимо было писать программы типа «блюдо спагетти».

Часто результатом трешинга является снижение производительности. Например, возможен такой сценарий развития событий. При глобальном алгоритме замещения, процесс, которому не хватает кадров, начинает отбирать их у других процессов, а те в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти, а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Планировщик процессов видит это и увеличивает степень мультипрограммирования, ещё более ухудшая ситуацию. Таким образом, пропускная способность системы падает из-за трешинга.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть уменьшен за счет использования локальных алгоритмов замещения. Если даже один из процессов попал в трешинг, это не сказывается на других процессах. Однако, он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Для предотвращения трешинга нужно выделить процессу столько кадров, сколько ему нужно для **оптимальной работы**. Для этого нужно узнать, сколько ему нужно. Например, можно попытаться выяснить, как много страниц процесс реально использует. Этот подход определяет *модель локальности* выполнения процесса.

Модель локальности состоит в том, что когда процесс выполняется, он двигается от одной локальности к другой. Локальность - набор страниц, которые активно используются вместе. Программа обычно состоит из нескольких различных локальностей, которые могут перекрываться. Например, когда вызвана процедура, она определяет новую локальность, состоящую из инструкций процедуры, ее локальных переменных и множества глобальных переменных. После ее завершения процесс оставляет эту локальность, но может вернуться к ней вновь. Таким образом, локальность определяется кодом и данными программы.

Если процессу выделять меньше кадров, чем ему нужно для поддержки его локальности он будет находиться в состоянии трешинга. Следовательно, рабочее множество процесса должно

включать достаточное количество кадров, чтобы минимизировать трешинг. Отсюда следует наиболее важное свойство рабочего множества — размер. И отсюда же следует метод борьбы с трешингом: если рабочие множества процессов не помещаются в память и начинается трешинг, то один из процессов надо попридержать. Также для увеличения производительности системы используются алгоритмы динамической настройки рабочих множеств.

1.7. Ввод/вывод в операционной системе

1.7.1. Устройства ввода/вывода

Управление устройствами ввода-вывода компьютера относится к управлению ресурсами — вторая основная функция операционной системы (см. п. 1.2). Она должна передавать устройствам команды, перехватывать прерывания и обрабатывать ошибки. Кроме того, ОС также должна обеспечивать интерфейс между устройствами и остальной частью вычислительной системы. В целях программирования этот интерфейс должен быть одинаковым для всех типов устройств (предоставление процессам «абстрактной» виртуальной машины — первая основная функция ОС).

Устройства, которые могут взаимодействовать с компьютером (а точнее, с процессором и памятью компьютера) весьма разнообразны: таймер, жёсткие диски, дисплеи, клавиатура, мышь, модемы, принтеры, сканеры, фотоаппараты и т.д., вплоть до весьма экзотических устройств специализированных вычислительных систем. Часть этих устройств может быть встроена внутрь корпуса компьютера и взаимодействовать с процессором и памятью по внутренним информационным магистралям (шинам) компьютера, а часть находиться за пределами и общаться с компьютером через различные линии связи. Однако, несмотря на всё многообразие устройств, управление их работой и обмен информацией с ними строятся на относительно небольшом количестве принципов.

Устройства ввода-вывода делятся на два типа: *блок-ориентированные* устройства и *байт-ориентированные* устройства. Блок-ориентированные устройства хранят информацию в блоках фиксированного размера, каждый из которых имеет свой собственный адрес (например, порядковый номер блока). Самое распространенное блок-ориентированное устройство - диск. В байт-ориентированных устройствах информация неадресуема и потому они не позволяют производить операцию поиска, они только генерируют и/или только потребляют последовательность байтов. Примерами являются терминалы, строчные принтеры, сетевые адаптеры, мыши (для указания позиции на экране), крысы (для лабораторных экспериментов по психологии) [18] и др.

Такая схема классификации несовершенна. Некоторые внешние устройства не относятся ни к одному из этих двух классов, например, часы, которые, с одной стороны, не адресуемы, а с другой стороны, не порождают потока байтов. Это устройство только выдает сигнал прерывания в

некоторые моменты времени. Обычно поступают так: если устройство точно блочное, то оно считается *блок-ориентированным,* а если символьное или не очень ясно какое, то *байт-ориентированным.* Такое разделение на блочные и символьные устройства является достаточно всеобъемлющим и неплохо подходит в качестве основы, позволяющей добиться того, чтобы программное обеспечение операционных систем не зависело от устройств ввода-вывода.

Внешнее устройство обычно состоит из механического и электронного компонента. В этом случае электронный компонент называется *контроллером* устройства или адаптером, а механический компонент представляет собой собственно устройство. Существует достаточно много устройств, состоящих только из электронного компонента, например, сетевые или аудио карты. Такие устройства (которые имеют встроенный контроллер) называются интеллектуальными. При этом контроллер представляет собой небольшую ЭВМ, в составе которой специализированный процессор, память оперативная, память ПЗУ (для хранения внутренних программ устройства), датчики и интерфейсы для управления электро-механической частью устройства.

Также существуют устройства, в составе которых нет электронного компонента, но их иногда тоже нужно подключать к компьютеру, например, дисковод floppy. Такие устройства называются «тупыми». В этом случае контроллер для таких устройств является составной частью компьютера. Например, для управления дисководами floppy на системной плате ПЭВМ устанавливается специальный контроллер. Другой пример: софт-принтеры, софт-модемы, софт-сканеры и др. устройства, которые фирмы-производители выпускают с неполной электронной частью (не установленной ради экономии), а те алгоритмы, что должна выполнять электроника, реализуют в драйвере устройства.

Если устройство электронно-механическое, то операционная система имеет дело не с устройством, а с контроллером этого устройства. Контроллер, как правило, выполняет простые функции, например, преобразует поток бит в блоки, состоящие из байт, и осуществляют контроль и исправление ошибок. Каждый контроллер имеет несколько регистров, которые используются для взаимодействия с центральным процессором — до четырёх: они называются регистрами ***состояния***, ***управления***, ***входных данных*** и ***выходных данных***. Иногда они объединяются, например регистры состояния и управления — в один регистр статуса, а регистры входных и выходных данных в один регистр ввода-вывода.

В некоторых компьютерах эти регистры являются частью физического адресного пространства (оперативной памяти). В таких компьютерах нет специальных операций ввода-вывода, и обмен с ними ничем не отличается от действий, производимых для передачи информации между оперативной памятью и процессором.

В других компьютерах адреса регистров ввода-вывода, называемых часто портами, образуют собственное адресное пространство за счет введения специальных операций ввода-

вывода (например, команд IN и OUT в процессорах i86) и тогда процесс обмена информацией инициируется специальными командами ввода-вывода и включает в себя несколько другие действия, связанные с переключением адресных пространств, инициацией обмена уже в пространстве ввода-вывода, а по завершению — переключением обратно в адресное пространство оперативной памяти.

Операционная система выполняет ввод-вывод, записывая команды в регистры контроллера. Например, контроллер floppy принимает 15 команд, таких как READ, WRITE, SEEK, FORMAT и т.д. Когда команда принята, операционная система оставляет контроллер и занимается другой работой, например, распределяет процессор какому-либо прикладному процессу, а контроллер в это время организует выполнение устройством команды. После того, как выполнение команды будет завершено, контроллер устройства организует прерывание («аппаратное») процессора для того, чтобы передать управление процессором операционной системе. В операционной системе по прерыванию инициируется обработчик этого прерывания, который должен проверить результаты операции, то есть, проверить статус устройства и, если надо, то получить результаты, прочитав информацию из регистров контроллера, опять же с переключением в пространство ввода-вывода и обратно. И т. д.

1.7.2. Организация программного обеспечения ввода/вывода

Основная идея организации программного обеспечения ввода-вывода состоит в разбиении его на несколько уровней, причем нижние уровни обеспечивают экранирование особенностей аппаратуры от верхних, а те, в свою очередь, обеспечивают удобный интерфейс для пользователей.

Ключевым принципом является независимость от устройств. Вид программы не должен зависеть от того, читает ли она данные с flash-диска, с гибкого диска, с жесткого диска, из pipe или из socket'a — везде должен работать один и тот же системный вызов, например read().

Очень близкой к идее независимости от устройств является идея единообразного именования устройств, то есть, для именования устройств в операционной системе должны быть приняты единые правила, например, с использованием файлов каталога dev, а после открытия файла устройства — дескриптора файла.

Другим важным вопросом для программного обеспечения ввода-вывода является обработка ошибок, возникающих при работе устройства. Очевидно, что ошибки следует обрабатывать как можно быстрее и, следовательно, как можно ближе к аппаратуре. Если контроллер обнаруживает ошибку чтения, то он должен попытаться ее скорректировать. Если же это ему не удается, то исправлением ошибок должен заняться драйвер устройства. Многие ошибки могут исчезать при

повторных попытках выполнения операций ввода-вывода, например, ошибки, вызванные наводками на кабель при передаче по сети. И только если нижний уровень не может справиться с ошибкой, он сообщает об ошибке верхнему уровню.

Еще один ключевой вопрос — способ передачи данных: синхронный (блокирующий) против асинхронного (управляемого прерываниями). Большинство операций физического ввода-вывода выполняется асинхронно — операционная система начинает передачу и переключает процессор на другую работу, пока не наступает прерывание. Однако, пользовательские программы намного легче писать, если операции ввода-вывода блокирующие (синхронные) - после команды READ программа автоматически приостанавливается до тех пор, пока данные не попадут в буфер программы. Преобразование асинхронных операций ввода-вывода в синхронное представление данных пользовательским программам выполняет операционная система, конкретно это делает «Планировщик задач» при обработке системного вызова прикладной программы.

Большое значение имеет буферизация данных, так как часто данные, поступающие с устройства, не могут быть сразу переданы программам. Например, когда пакет приходит по сети, операционная система не знает, кому его передать, пока не изучит его содержимое, поэтому этот пакет нужно где-то временно пристроить — в некотором временном буфере. Буферизация подразумевает копирование данных в значительных количествах из одной области памяти в другую область памяти пока данные не достигнут потребителя, что часто является основным фактором снижения производительности операций ввода-вывода.

И наконец, последняя проблема состоит в том, что одни устройства могут быть разделяемыми, а другие — выделенными. Вообще говоря, практически все устройства (даже более того, практически все ресурсы) изначально (с момента изобретения) являлись выделенными. Но некоторые устройства уже практически сразу снабжались средствами разделения доступа к ним («расшаривания»), а для других такие средства появились не сразу и потому появилось такое условное деление устройств на «разделяемые» и «выделенные». Например, диски практически сразу были снабжены файловыми системами как средством разделения доступа и потому про диски всегда говорят, что это разделяемые устройства. А, вот, для принтеров такие средства появились не сразу, а только с появлением многозадачных многопользовательских операционных систем, и потому сложилось мнение, что принтеры - это выделенные устройства.

Универсальные операционные системы, которые мы чаще всего эксплуатируем сейчас, являются многозадачными, а некоторые (Unix/Linux) и многопользовательскими, то есть, в вычислительной системе одновременно выполняются много программ, и потому нормальным является возникновение ситуаций, когда одновременно несколько программ (процессов) желают поиметь доступ к некоторому устройству. Следовательно, устройства должны правильно «расшариваться». И вот здесь операционная система должна вставить свои весомые «пять копеек»: все попытки доступа к устройству должны быть перехвачены и организованы посредством

выстраивания в очередь запросов к устройству. Да и сам запрос — непосредственную работу с устройством — реализует сама операционная система с помощью драйвера устройства.

Но пользователь, незнакомый со структурой операционной системы, не видит модуля операционной системы — посредника между ним и устройством, который как раз и реализует «разделяемость» (расшариваемость) устройства. Для дисков таким модулем является драйвер файловой системы, который расположен «поверх» драйвера самого устройства, и он используется практически всегда, даже в однозадачных операционных системах. А, вот, для принтеров такого модуля, обеспечивающего «разделяемость» принтера может и не быть (в однозадачных операционных системах, а иногда и в многозадачных), а есть (а иногда и его нет) только драйвер самого устройства — принтера. И тогда создаётся впечатление, что диск - «разделяемый» ресурс, а принтер - «выделенный».

На такое разделение устройств на две группы влияет также то, используется ли устройство самой операционной системой. Например, диск, таймер, дисплей — используются и потому для них в составе операционной системы наличествуют средства разделения доступа, а, вот, принтеры, модемы, веб-камеры операционной системой не используются и для них средства «расшаривания» появляются постепенно с развитием технологий.

В некоторых вычислительных системах, в случае отсутствия на соответствующем уровне операционной системы модуля, обеспечивающего «разделяемость» устройства, для решения задачи «разделения» («расшаривания»), запускается процесс на пользовательском уровне — сервис, реализуемый с помощью программы специального типа — демона, который и решает эту задачу управления доступом к устройству. Например, в современных Unix/Linux таким сервисом для принтеров является сервис cups.

Для решения поставленных проблем обычно программное обеспечение ввода-вывода делится на четыре слоя (рис. 19):

4 слой: Пользовательский слой программного обеспечения.

3 слой: Независимый от устройств слой операционной системы, обеспечение «разделяемости».

2 слой: Драйверы устройств.

1 слой: Обработка прерываний.

Рис. 19. Уровни программного обеспечения ввода/вывода

1.7.3. Обработка прерываний

Аппаратное прерывание (это всегда прерывание от устройства) обрабатывается модулем операционной системы - процедурой обработки прерывания. В тех случаях, когда операционная система написана на языке Си, этот модуль оформляется как функция языка Си и является частью драйвера устройства.

С помощью механизма прерываний реализуется асинхронный метод взаимодействия операционной системы с устройством. Но процесс, инициировавший операцию ввода-вывода, работает с операционной системой синхронно: он блокируется (останавливается, переводится в состояние «ожидания») до завершения операции. Операционная система с помощью драйвера устройства транслирует запрос ввода/вывода в команду для устройства, записывая в регистр управления устройства необходимые биты (код команды) а в регистры ввода/вывода параметры команды или данные, если они необходимы. После завершения этого операционная система переключается на другую работу, запуская с помощью планировщика какой-либо другой процесс

(чтобы процессор не простаивал).

Когда устройство выполнит команду, оно выставит в своём регистре состояния специальный бит прерывания, который отдельной линией подключен к микросхеме арбитра прерываний. То, как подключена эта линия к арбитру прерываний, определяет номер прерывания. Арбитр прерываний определит, возможно ли прерывание (не «замаскировано» ли) и от кого оно и, если это возможно, прервёт работу процессора. Процессор остановит текущий процесс, сохранит его состояние в PCB, по таблице прерываний (входом в которую является номер прерывания) определит адрес процедуры обработки этого прерывания и переключится на выполнение этой процедуры.

Процедура обработки прерывания должна прежде всего проверить регистр состояния устройства — не произошло ли ошибки при выполнении команды. Если ошибки нет, тогда процедура обработки прерывания должна выполнить срочные действия, которые требуются для завершения инициированной операции ввода/вывода и завершиться. Как правило, в начале процедуры стоит команда cli, запрещающая другие прерывания («маскирующая» прерывания) на время выполнения данной процедуры, а в конце команда sti, снова разрешающая прерывания. Это обусловлено тем, что процедура обработки прерывания является «критической секцией», то есть, участком кода, который не должен прерываться, иначе могут быть утеря ны данные. Далее драйвер устройства должен выполнить остальные действия, определённые алгоритмом взаимодействия с устройством для инициированного запроса, например, подготовить сообщение для процесса, который этот запрос сделал.

По завершении работы драйвера операционная система, в лице планировщика заданий, должна обработать эту ситуацию по «вновь открывшимся обстоятельствам», например, перевести процесс из состояния «ожидания» в состояние «готовность», поместив ссылку на его PCB в соответствующую очередь планировщика. То есть, результатом прерывания будет то, что ранее заблокированный процесс теперь сможет продолжить свое выполнение.

Альтернативой этого алгоритма работы с устройством, является метод «опроса», который является синхронным методом работы с устройством. Он заключается в том, что драйвер, после записи команды запроса в регистр управления устройства, не завершается и управление не передаётся планировщику заданий. Драйвер устройства продолжает работать, в цикле опрашивая регистр состояния устройства, до появления в нём либо кода ошибки выполнения команды, либо кода успешного завершения команды. Когда эта информация появится, драйвер и операционная система далее отработают также как в предыдущем случае. То есть, в этом методе в составе драйвера отсутствует процедура обработки прерываний и процессор не переключается на время выполнения устройством команды на другую работу. Очевидно, что это приводит к снижению эффективности работы вычислительной системы. Однако, такой метод вполне допустим и нередко используется в однозадачных операционных системах, поскольку проще программируется.

1.7.4. Драйверы устройств

Весь зависимый от устройства код помещается в драйвер устройства. Каждый драйвер управляет устройствами одного типа или, может быть, одного класса.

В операционной системе именно драйвер устройства знает о конкретных особенностях какого-либо устройства. Например, только драйвер диска имеет дело с командами процессора контроллера диска (так называемыми, «канальными» командами), знает временные характеристики выполнения команд контроллера и их параметры, умеет работать с буфером диска и другие факторы, обеспечивающие правильную работу диска. Или, например, только драйвер принтера знает про язык взаимодействия с принтером (PCL для принтеров HP, язык esc-последвательностей для принтеров корейского или японского производства или язык Postscript).

Драйвер устройства через посредство других модулей операционной системы, принимает запрос от пользовательских программ и решает, как его выполнить. Типичным запросом является чтение n блоков данных. Если драйвер был свободен во время поступления запроса, то он начинает выполнять запрос немедленно. Если же он был занят обслуживанием другого запроса, то вновь поступивший запрос присоединяется к очереди уже имеющихся запросов, и он будет выполнен, когда наступит его очередь.

Первый шаг в реализации запроса ввода-вывода, например, для диска, состоит в преобразовании его из абстрактной формы в конкретную. Для дискового драйвера это означает вычисление номеров блоков данных (порядковых номеров блоков), формирование команды (преобразование запроса в код команды контроллера диска), запись команды и необходимых данных в регистры контроллера, инициация команды (указание контроллеру начать выполнение команды).

Контроллер, зная соответствие между порядковыми номерами блоков и их физическими адресами расположения на диске, преобразует номера блоков в номера цилиндров, головок, секторов, проверяет, где находятся в настоящий момент головки, организует перемещение головок на нужный цилиндр с помощью обработки информации с серво-дорожек, считывает информационную дорожку, выделяет из неё серво-метки, вычисляет с их помощью скорость вращения шпиндельного двигателя и корректирует эту скорость включением/отключением шпиндельного двигателя. Если поступившая команда является командой чтения, то раскодирует остальную часть информации с информационной дорожки, выделяя из неё отдельные блоки (сектора) и переписывает блоки в выходной буфер диска. Если команда была на запись, то наоборот, считывает блоки из входного буфера, кодирует их некоторым методом избыточного кодирования и пишет на дорожку. В конце выполнения команды на регистре состояния появляется код, свидетельствующий о успехах выполнения команды.

Короче говоря, драйвер решает на основании обработки очереди запросов, какие операции контроллер должен выполнить. Однако непосредственно работает с диском контроллер диска. Информация о том, где, на каком цилиндре, дорожке, секторе размещается информация, в каком виде хранится, контроллер драйверу не сообщает.

После передачи команды контроллеру драйвер должен решить, блокировать ли себя до окончания заданной операции или нет. Если операция занимает значительное время, как при чтении/записи некоторого блока данных, то драйвер прекращает работу и управление передаётся планировщику задач до тех пор, пока операция не завершится, тогда контроллер выставит прерывание и обработчик прерывания снова инициирует драйвер. Если команда ввода-вывода выполняется быстро (например, определение текущего состояния), то драйвер ожидает ее завершения без блокирования.

В Linux драйвер выглядит как набор взаимосвязанных функций, причём некоторые имеют частично предопределённые имена.

1.7.5. Аппаратно-независимый слой операционной системы

Большая часть программного обеспечения ввода-вывода является независимой от устройств. Точная граница между драйверами и независимым от устройств кодом определяется системой, так как некоторые функции, которые могут быть реализованы независимым способом, в действительности выполнены в виде драйверов для повышения эффективности или по другим причинам. Например, выше драйверов устройств работают драйверы файловых систем, реализуя логический уровень работы с устройствами, хотя такого устройства - «файловая система» - не существует.

Типичными функциями для аппаратно-независимого от устройств слоя являются:

- обеспечение общего интерфейса к драйверам устройств,
- именование устройств,
- защита устройств, в том числе, разграничение доступа к ним,
- обеспечение независимого размера блока,
- буферизация,
- распределение памяти на блок-ориентированных устройствах,
- распределение и освобождение выделенных устройств,
- уведомление об ошибках.

Верхним слоям программного обеспечения не удобно работать с блоками разной величины, поэтому данный слой обеспечивает единый размер блока, например, за счет объединения нескольких различных блоков в единый логический блок. В связи с этим верхние уровни имеют

дело с абстрактными устройствами, которые используют единый размер логического блока независимо от размера физического сектора на конкретном устройстве. Пример: «блок» файловой системы в Unix/Linux эквивалентное понятию «кластер» в Windows.

При создании файла или заполнении его новыми данными необходимо выделить ему новые блоки. Для этого ОС должна вести список или битовую карту свободных блоков диска. На основании информации о наличии свободного места на диске может быть разработан алгоритм поиска свободного блока, независимый от устройства и реализуемый программным слоем, находящимся выше слоя драйверов устройств, например, драйвером файловой системы.

С помощью файлов решается задача разграничения доступа к устройству разных программ — каждая программа работает со своими файлами, независимо от других, со своей именованной частью диска. Тем самым, драйвер файловой системы обеспечивает «расшаривание» диска между разными программами и, соответственно, разными пользователями.

Буферизация является важной как для блочных, так и для символьных устройств. Для блочных устройств аппаратное обеспечение обычно требует, чтобы чтение или запись производились большими блоками. Однако пользовательские программы вправе передавать любые объемы информации. Поэтому если программа передает только половину блока, операционная система обычно не сразу записывает эти данные на диск, а дожидается передачи оставшейся части блока. Что же касается символьных устройств, то программа может передавать данные быстрее, чем устройство в состоянии их воспринять, следовательно, и здесь нужна буферизация. Или, например, данные, поступающие от клавиатуры, могут опережать считывание их программой, значит и здесь не обойтись без буфера.

Некоторые устройства, например привод CD-RW, рассчитаны на монопольное владение процессом (например, программой cdrecord при записи «болванки») в течении некоторого времени. Операционная система должна рассмотреть запросы от других программ на использование такого устройства и, если устройство занято, то отказать в выполнении запроса.

1.7.6. Пользовательский слой программного обеспечения

Хотя большая часть программного обеспечения ввода-вывода находится внутри операционной системы, некоторая его часть содержится в библиотеках, связываемых с пользовательскими программами. Системные вызовы, включая вызовы ввода-вывода, обычно оформляются как библиотечные функции.

Если программа, написанная на языке Си, содержит вызов

count = write (fd, buffer, nbytes),

то библиотечная функция write будет связана с программой.

Набор подобных функций является частью системы ввода-вывода и определен в стандарте POSIX, то есть, это библиотечные функции, соответствующие системным вызовам, их параметры, что они должны делать и какой результат возвращать. В частности, форматирование ввода или вывода выполняется библиотечными функциями, например функция printf языка Си принимает строку формата и, возможно, некоторые переменные в качестве входной информации, затем строит строку символов ASCII и делает системный вызов write (возможно, не один) для вывода этой строки. Стандартная библиотека libc языка Си содержит большое число функций, которые выполняют ввод-вывод и работают как часть пользовательской программы.

Другой категорией программного обеспечения ввода-вывода является подсистема спулинга (spooling). Спулинг - это способ работы с выделенными устройствами в мультипрограммной системе. Примером типичного устройства, требующего спулинга, является принтер. Хотя технически легко позволить каждому пользовательскому процессу открыть специальный файл, связанный с принтером (изменив права доступа к этому файлу), такой способ опасен из-за того, что пользовательский процесс может монополизировать принтер на произвольное время. Вместо этого создается специальный процесс - монитор, который получает исключительные права на использование этого устройства. Также создается специальный каталог, называемый каталогом спулинга. Для того, чтобы напечатать файл, пользовательский процесс помещает этот файл в каталог спулинга. Процесс-монитор по очереди распечатывает все файлы, содержащиеся в каталоге спулинга. Так работают сервисы печати lpr и cups.

С помощью подсистемы спулинга реализуется «расшаривание» выделенного устройства на пользовательском уровне (вне пределов операционной системы). Аналогичным образом «расшариваются» не только реальные устройства, но и чисто программные сервисы, например доступ к почтовой системе: для отправки письмо помещается в очередь почтового сервера, а он, обнаружив, что очередь не пуста, организует его отправку адресату.

То есть, такой способ «расшаривания» устройств и функций вычислительной системы — с помощью запуска специальных программ-демонов, которые реализуют сервисы в вычислительной системе — реализуется на пользовательском уровне. В этом случае пользовательский процесс обращается к программе-демону по общеизвестному адресу этого демона в вычислительной системе, передаёт ему запрос и данные для выполнения запроса, а демон организует синхронное или асинхронное выполнение этого запроса, возможно, некоторым устройством (например, так работает сервис логирования syslog).

1.7.7. Особенности работы с файлами

Информация о файлах, используемых процессом, входит в состав его контекста и сохраняется в его блоке управления - PCB. В операционных системах Unix/Linux информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, называющемся таблицей открытых файлов или таблицей файловых дескрипторов. В строке этой таблицы располагается информация об открытом файле: что за файл, на каком устройстве, некоторые атрибуты файла, режим работы с файлом (только чтение, только запись, чтение/запись) и др. Индекс элемента этого массива (номер строки этой таблицы), соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока.

Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное число, которое для текущего процесса в текущий момент времени однозначно определяет некоторый действующий канал ввода-вывода. Первые три строки этой таблицы уже на этапе старта любой программы заняты: они ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор 0 соответсвует стандартному потоку ввода (stdin), файловый дескриптор 1 - стандартному потоку вывода (stdout), файловый дескриптор 2 - стандартному потоку для вывода ошибок (stderr). В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок - с текущим терминалом.

Файловый дескриптор используется в качестве параметра, описывающего поток ввода-вывода, для системных вызовов, выполняющих операции над этим потоком. Поэтому, прежде чем совершать операции чтения данных из файла и записи их в файл, необходимо поместить информацию о файле в таблицу открытых файлов и определить соответствующий файловый дескриптор. Для этого применяется процедура открытия файла, осуществляемая системным вызовом open().

Системный вызов open() использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент открытия файла. Особенно часто используются флаги **O_RDONLY**, **O_WRONLY**, **O_RDWR, O_CREAT** и **O_EXCL**. Флаги **O_RDONLY**, **O_WRONLY**, **O_RDWR** являются взаимоисключающими: хотя бы один из них должен быть употреблен и наличие одного из них не допускает наличия двух других. Эти флаги описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в последующем: только чтение, только запись, чтение и запись.

Кроме того, у каждого файла существуют атрибуты прав доступа для различных категорий пользователей. Если файл с заданным именем существует на диске, и права доступа к нему для пользователя, от имени которого работает текущий процесс, не противоречат запрошенному набору операций, то операционная система сканирует таблицу открытых файлов от ее начала к концу в поисках первого свободного элемента, заполняет его и возвращает индекс этого элемента (номер строки) в качестве файлового дескриптора открытого файла. Если файла на диске нет, не хватает прав или отсутствует свободное место в таблице открытых файлов, то констатируется возникновение ошибки.

В случае если допускается, что файл на диске может отсутствовать, и нужно, чтобы он тогда был создан, флаг для набора операций должен использоваться в комбинации с флагом **O_CREAT**. Если файл существует, то все происходит по рассмотренному выше сценарию. Если файла нет - то сначала выполняется создание файла с набором прав, указанном в параметрах системного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться (как, например, в Linux).

В случае, когда требуется, чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами **O_CREAT** и **O_EXCL**.

Существуют и другие флаги.

В языке программирования Си имеются функции работы с файлами из стандартной библиотеки ввода-вывода такие, как fopen(), fread(), fwrite(), fprintf(), fscanf(), fgets() и т.д. Эти функции входят как неотъемлемая часть в стандарт ANSI на язык С и позволяют программисту получать информацию из файла или записывать ее в файл при условии, что программист обладает определенными знаниями о содержимом передаваемых данных. Так, например, функция fgets() используется для ввода из файла последовательности символов, заканчивающейся символом '\n' - перевод каретки; функция fscanf() производит ввод информации, соответствующей заданному формату, и т. д.

С точки зрения потоковой модели операции, определяемые функциями стандартной библиотеки ввода-вывода, не являются потоковыми операциями, так как каждая из них требует наличия некоторой структуры передаваемых данных, которую программист должен предварительно определить в своей программе. В операционных системах Unix/Linux эти функции представляют собой надстройку - сервисный интерфейс - над системными вызовами, осуществляющими прямые потоковые операции обмена информацией между процессом и файлом и не требуюшими никаких априорных знаний о структуре этой информации - системные вызовы read() и write().

После завершения потоковых операций, процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный досброс буферов на линии

связи, освободятся выделенные ресурсы операционной системы, и элемент таблицы открытых файлов, соответствующий файловому дескриптору, будет отмечен как свободный. За эти действия отвечает системный вызов close(). Надо отметить, что при завершении работы процесса с помощью явного или неявного вызова функции exit() происходит автоматическое закрытие всех открытых потоков ввода-вывода.

1.8. Оболочки операционной системы

1.8.1. Определение оболочки

Оболочка операционной системы (*shell* - «оболочка») — это интерпретатор команд операционной системы, обеспечивающий интерфейс для взаимодействия пользователя с функциями системы. Оболочка необходима, поскольку напрямую (прямо с клавиатуры или мышкой вызвать функцию (функцию на языке Си, на котором написана операционная система), передать ей параметры и заставить что-то сделать никак нельзя.

В общем случае, различают оболочки с двумя типами интерфейса для взаимодействия с пользователем: текстовый пользовательский интерфейс (TUI) и графический пользовательский интерфейс (GUI). Графические оболочки требуют для своей работы всегда гораздо большей вычислительной мощности (в десятки раз) по сравнению с текстовыми командными оболочками.

Оболочку с текстовым пользовательским интерфейсом называют также командным интерпретатором, который используют для обеспечения интерфейса командной строки в операционных системах. Командные интерпретаторы могут представлять собой самостоятельные языки программирования, с собственным синтаксисом и отличительными функциональными возможностями.

В операционные системы MS-DOS и Windows 9x включён командный интерпретатор, command.com, в Windows NT включён cmd.exe, начиная с Windows XP (пакет обновления 2) доступен PowerShell, который является встроенным компонентом ОС, начиная с Windows 7 и Windows 2008 Server.

В Unix-подобных системах у пользователя есть возможность менять командный интерпретатор, используемый по умолчанию. Из командных оболочек Unix наиболее популярны bash, csh, ksh, zsh.

1.8.2. Функции оболочек

Командный интерпретатор исполняет команды своего входного языка, заданные в командной строке или поступающие из стандартного ввода или указанного файла.

В качестве команд интерпретируются вызовы системных или прикладных утилит, а также

управляющие конструкции (см. рис. 20). То есть, командой оболочки («командой в Linux») является либо имя некоторой утилиты, либо оператор языка shell. Кроме того, оболочка отвечает за раскрытие шаблонов имен файлов и за перенаправление и связывание ввода-вывода утилит. Основной набор этих утилит (около ста) описывается стандартом POSIX 1003.2, но кроме них в разных версиях системы Unix/Linux могут существовать дополнительные программы. В состав этого набора входят командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами.

В совокупности с набором утилит, оболочка представляет собой операционную среду, язык программирования и средство решения как системных, так и некоторых прикладных задач, в особенности, автоматизации часто выполняемых последовательностей команд.

```
while (1)
  {                              // начало основного цикла
    write (1, "$ ", 2);          // выводим приглашение
    readcmd (cmd, args);         // смотрим, что ввёл пользователь
    if ((pid = fork ()) == 0) // создаём подпроцесс и если мы в нём,
      {
        exec (cmd, args, 0); // то запускем указанную программу
      }
    else
    if (pid > 0)                 // если же мы в родительском
      {
        wait (0);                // ждём завершения запущенной проги
      }
    else                         // иначе ошибка
      {
        perror ("Error on fork\n");
      }
  }
```

Рис. 20. Скелет оболочки.

Стандартом POSIX (ISO/IEC 9945 - Том 3. Оболочка и утилиты) определён язык оболочки, включающий конструкции последовательного (перевод строки, точка с запятой), условного (if, case, ||, &&) и циклического (for, for in, while, until) исполнения команд, а также оператор присваивания.

Стандартом также определён режим редактирования вводимых команд, являющийся подмножеством команд стандартного текстового редактора (**vi**).

У многих версий системы Unix имеется графический интерфейс пользователя, схожий с популярными интерфейсами, применёнными на компьютере Macintosh и впоследствии в системе Windows. Однако многие до сих пор предпочитают интерфейс командной строки, называемый оболочкой (shell). Подобный интерфейс значительно быстрее в использовании, существенно мощнее и проще расширяется.

1.8.3. Командная оболочка Баурна (bash)

Когда оболочка запускается, она инициализируется, а затем печатает на экране символ приглашения к вводу (обычно это знак доллара или процента) и ждет, когда пользователь введет командную строку. После того как пользователь введет командную строку, оболочка извлекает из нее первое слово и ищет файл с таким именем. Если такой файл удается найти, оболочка запускает его. При этом работа оболочки приостанавливается на время работы запущенной программы. По завершении работы программы оболочка снова печатает приглашение и ждет ввода следующей строки.

Здесь важно подчеркнуть, что оболочка представляет собой обычную пользовательскую программу. Все, что ей нужно, – это способность ввода с терминала и вывода на терминал, а также возможность запускать другие программы (см. man bash).

Синтаксис команд. При разборе введённой пользователем команды используются следующие понятия:

- пробел или табуляция — интерпретируется как **разделитель;**

- слово (лексема) - последовательность символов, рассматриваемая командным интерпретатором как единое целое;

- имя (идентификатор) - слово, состоящее только из алфавитноцифровых символов и символов подчеркивания, и начинающееся с буквы или символа подчеркивания;

- метасимвол - символ, разделяющий слова, если он не замаскирован; это один из следующих символов: |, &, ;, (,), <, >, пробел, табуляция;

- управляющий оператор - лексема, выполняющая функцию управления; это один из следующих символов: ||, &, &&, ;, ;;, (,), |, <перевод строки>.

Введённая пользователем командная строка может интерпретироваться как:

- простая команда - это некоторое слово (лексема), которое задает команду, которую надо выполнить; оставшиеся слова передаются как аргументы вызванной команде; возвращаемым значением простой команды является ее статус выхода (код завершения);

- конвейер - это последовательность одной или более команд, разделенных символом | (вертикальная черта); при этом, стандартный выходной поток предыдущей команды связывается со стандартным входным потоком последующей команды;

- список - это последовательность одного или более конвейеров, разделенных одним из операторов ;, &, && или ||, и не обязательно завершающаяся одним из операторов ;, & или <перевод строки>; если команда завершается управляющим оператором &, интерпретатор выпол-няет команду в фоновом режиме (отсоединённым от терминала) в порожденном интерпретаторе и в этом случае командный интерпретатор не ждет завершения команды, а

статус выхода в этом случае — 0; команды, разделенные выполняются последовательно, то есть, командный интерпре-татор ждет поочередно завершения каждой из команд, и статус возврата списка в этом случае совпадает со статусом возврата последней выполненной команды; управляющие операторы && и || обозначают условные списки, соответственно, И-списки и ИЛИ-списки;

- составные команды — это команды, составленные из всего вышеперечисленного, а также могут содержать выражения и целые программы (скрипты), составленные на языке shell.

Выполнение команды. После разбиения команды на слова, если в результате получилась простая команда с необязательным списком аргументов, выполняются следующие действия:

- командный интерпретатор пытается найти команду; прежде всего, если существует функция командного интерпретатора с таким именем, она вызывается и аргументы команды передаются этой функции как параметры; иначе командный интерпретатор ищет команду в списке встроенных команд; если такая встроенная команда есть, она выполняется; иначе, если имя не является именем функции или именем встроенной команды и не содержит пробелы, командный интерпретатор bash просматривает каждый каталог в значении переменной PATH в поисках выполняемого файла с соответствующим именем; если команда при таком поиске не найдена, командный интерпретатор выдает соответствующее сообщение и завершает выполнение команды;

- если команда найдена или имя команды содержит косые черты («слэши», то есть, задан путь к некоторой программе), командный интерпретатор выполняет соответствующую команду в отдельной среде выполнения; при этом, аргумент 0 устанавливается равным имени команды, и ей передаются параметры, соответствующие аргументам в командной строке, если они заданы;

- если выполнить команду не удалось, потому что файл не соответствует поддерживаемым выполняемым форматам, и если этот файл не является каталогом, то предполагается, что файл является сценарием командного интерпретатора, содержащим его команды; в этом случае файл просматривается и если файл текстовый и начинается с #!, то остаток первой строки задает интерпретатор для программы; командный интерпретатор запускает указанный интерпретатор и этому интерпретатору в качестве аргументов передается один необязательный аргумент, затем имя интерпретатора из первой строки программы, затем имя самой программы и ее аргументы, если они заданы.

Стандартно программа не должна открывать терминал, чтобы прочитать с него или вывести на него строку. Вместо этого запускаемые программы автоматически получают доступ к файлу, называемому стандартным устройством ввода (standard input), к файлу, называемому стандартным устройством вывода (standard output), а также к файлу, называемому стандартным устройством для вывода сообщений об ошибках (standard error). По умолчанию всем трем устройствам соответствует терминал, то есть клавиатура для ввода и экран для вывода.

Многие программы в системе Unix читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода. Стандартные ввод и вывод также можно перенаправить, что является очень полезным свойством. Для этого используются символы «<» и «>» соответственно. Разрешается их одновременное использование в одной командной строке. Программа, считывающая данные со стандартного устройства ввода, выполняющая определенную обработку этих данных и записывающая результат в поток стандартного вывода, называется *фильтром*.

В системе Unix часто используются командные строки, в которых первая программа в командной строке формирует вывод, используемый второй программой в качестве входа. Система Unix предоставляет более простой способ реализации этого механизма, который заключается в использовании вертикальной черты, называемой *символом канала*. Набор команд, соединенных символом канала, называется *конвейером* и может содержать произвольное количество команд.

Поскольку Unix является универсальной многозадачной системой, то пользователь может одновременно запустить несколько программ, каждую в виде отдельного процесса, например, в фоновом режиме. Для этого в конце команды указывается символ & (отсоединения от терминала). Конвейеры также могут выполняться в фоновом режиме. Можно одновременно запустить несколько фоновых конвейеров.

Список команд оболочки может быть помещен в файл, а затем этот файл с командами может быть выполнен, для чего нужно запустить оболочку с этим файлом в качестве входного аргумента. Вторая программа оболочки просто выполнит перечисленные в этом файле команды одну за другой, точно так же, как если бы эти команды вводились с клавиатуры. Файлы, содержащие команды оболочки, называются *сценариями оболочки* или *скриптами*. Сценарии оболочки могут присваивать значения переменным оболочки и затем считывать их. Они также могут запускаться с параметрами.

Таким образом, сценарии оболочки представляют собой настоящие программы, написанные на языке оболочки. Существует альтернативная оболочка csh, разработанная таким образом, чтобы сценарии оболочки (и команды языка вообще) выглядели во многих аспектах подобно программам на языке C.

1.8.4. Графические оболочки

Графические оболочки для Windows. Многие версии операционной системы Windows используют в качестве своей оболочки интегрированную среду Проводника Windows. Проводник Windows представляет собой визуальную среду управления, включающую в себя **один** *Рабочий стол, Меню Пуск, Панель задач*, а также функции управления файлами. Ранние версии Windows 3.xx в качестве графической оболочки включали менеджер программ.

Многие сторонние разработчики предлагают альтернативные среды, которые могут быть использованы вместо оболочки проводника, включенной по умолчанию компанией Microsoft в систему Windows.

Перечень оболочек для Microsoft Windows: Aston shell, BB4Win, Cairo, Chroma Emerge, Desktop Geoshell, KDE, Litestep, Windows Explorer, Microsoft Bob и др.

Графические оболочки для Unix/Linux. Ими являются KDE, GNOME, XFce, ROX Desktop, Blackbox и другие.

KDE (изначально проект назывался Kool Desktop Environment) — свободная среда рабочего стола для Unix-подобных операционных систем. Построена на основе кросс-платформенного инструментария разработки пользовательского интерфейса Qt. Работает преимущественно под Unix-подобными операционными системами, которые используют графическую подсистему X Window System. Новое поколение технологии KDE 4 частично работает на Microsoft Windows и Mac OS X.

GNOME является альтернативной графической оболочкой, также свободной. Построена на основе инструментария разработки пользовательского интерфейса gtk. В GNOME также, как и в KDE, имеется панель (для запуска приложений и отображения их состояния), рабочие столы (несколько штук, где могут быть размещены данные и приложения), набор стандартных инструментов и приложений для рабочих столов, а также набор соглашений, облегчающих совместную работу и согласованность приложений.

Другие графические оболочки (xfce, fvwm, blackbox, jwm, owm, olwm, cwm и ещё не меньше десятка) являются не столь сложными, как KDE и GNOME и, соответственно, требуют для своей работы меньшей вычислительной мощности.

1.9. Введение в системное программирование

1.9.1. Определения

В соответствии с определением [19], **системная программа (утилита)** — это программа, предназначенная для поддержания работоспособности Системы Обработки Информации (СОИ, вычислительной системы) или повышения эффективности её использования. То есть, в определении говорится о программе, предназначенной для поддержания работоспособности СОИ — а это операционная система, и о программе, предназначенной для повышения эффективности использования СОИ — об утилите.

Таким образом, **системное программирование** — это процесс разработки системных программ, управляющих и обслуживающих, то есть, операционной системы и утилит.

С другой стороны, по определению Гегеля, система — единое целое, состоящее из множества компонентов и множества связей между ними. Тогда **системное программирование** —

это разработка программ сложной структуры.

Эти два определения совершенно не противоречат друг другу, поскольку совокупность системного ПО — это весьма сложная, существенно иерархически упорядоченная и тесно взаимодействующая совокупность операционной системы и большого набора утилит, причём сама ОС Linux, как уже было сказано выше, является пожалуй самым большим программным проектом, когда либо разрабатывавшимся человечеством.

А как всем известно, ОС Linux почти вся написана на С и чуть-чуть на ассемблере. И, следовательно, системное программирование — это в основном программирование на С. Причём, если для разработки утилит используется «обычный» С (какому учат на занятиях), то для разработки операционной системы используется некоторое подмножество языка, не включающее, например, привычных системных вызовов и некоторых других расширений языка.

Такое положение (что системное программирование — это программирование на С) сложилось в основном за последние два десятилетия и связано с распространением Unix и Unix-подобных ОС, прежде всего Linux, которые написаны почти целиком на С. Ранее, в 60/70/80-ые годы положение было иным: операционные системы писались почти всегда на ассемблере соответствующего процессора и, соответственно, под системным программированием понимали разработку программ на этом языке.

Вы могли заметить, здесь ничего не говорится о Windows и системном программировании под эту операционную систему. Это связано с тем прискорбным фактом, что под Windows системное программирование существует почти исключительно только внутри MS, а вне MS программирование под Windows — только прикладное. Почему так? А потому что, если кому-либо (просто программисту или фирме) удаётся написать реально полезную системную программу для Windows, то:

- либо эта программа теряет актуальность с выходом новой версии Windows (MS включает соответствующую функциональность в состав ОС; яркий пример, нортоновские утилиты — где они? а, ведь, десять-пятнадцать лет назад о них знали все);

- либо фирма покупается MS и опять же соответствующая функциональность становится частью Windows.

Это и приводит к тому, что условий для развития системного программирования под Windows нет из-за монополизма MS.

1.9.2. Технологические особенности системного программирования

Основные инструменты системного программиста:

- текстовый редактор — необходим для ввода исходных текстов и другой документации;

- транслятор — необходим для преобразования исходного текста в команды ЭВМ;

- отладчик — необходим для отладки программы;

- другие вспомогательные средства, как то: профилировщики текста, трассировщики выполнения, трассировщики системных вызовов и др.

Как правило, в качестве транслятора в настоящее время используется gcc (GNU C Compieler). IDE (интегрированные среды разработки) используются нечасто и, если используются то: а) для разработки системного ПО (не операционной системы), б) только те IDE, которые позволяют разработку консольных программ. Эти ограничения обусловлены тем, что во-первых, для написания исходных текстов модулей ОС используется почти всегда только обычный текстовый редактор и весьма специфические средства отладки, а во-вторых, системное ПО для Unix/Linux — это консольные программы, именно они чаще всего и являются «командами» Unix/Linux.

Таким образом, технология разработки модулей ОС выглядит следующим образом. Как правило, работа осуществляется в текстовом режиме (уровень 3 — режим командной строки) и состоит из следующих шагов:

1) в текстовом редакторе пишется исходник модуля; созданный file.c и, может быть, file .h, помещаются в нужное место (в соответствующий каталог) исходников ядра;

2) с помощью текстового редактора исправляется make-файл в соответствующем каталоге исходников ядра, то есть, в make-файл добавляются строки, которые обеспечат трансляцию и последующую сборку разрабатываемого модуля с ядром;

3) запускается перетрансляция и пересборка ядра и initrd.img, если необходимо; новое ядро под именем myvmlinuz помещается в /boot; сюда же кладётся и новый myinitrd.img;

4) с помощью текстового редактора исправляется конфигурационный файл загрузчика - в него добавляется пункт

```
menuentry 'New My Kernel — debugged'
{
  . . .
  Linux /boot/myvmlinuz . . . параметры загрузки, если нужны
  initrd /boot/myinitrd.img
};
```

5) систему перезагружаем, смотрим результаты нашей плодотворной работы;

6) если результаты не наблюдаются, то возвращаемся на пункт 1, исправляем, что напортачили, и продолжаем работу дальше.

В итоге мы должны получить некоторый модуль ядра, который так и останется в соответствующем каталоге дерева исходников ядра, а его транслированный вместе с ядром образ

будет обеспечивать нужную нам функциональность ядра. Если мы хотим, чтобы наши усилия не канули в лету, а начали распространяться с некоторой очередной версией ядра, то необходимо обсудить результаты работы в соответствующей конференции по ядру в Интернет и отправить исходник и make-файл Л. Торвальдсу.

Технология разработки системных утилит немного отличается и выглядит следующим образом:

1) в текстовом редакторе пишется исходник программы, некоторые файлы file.h и file.c (возможно даже несколько файлов); файлы помешаются в некоторый каталог ~/myproga;

2) с помощью текстового редактора создаётся make-файл в каталоге myproga, то есть, в make-файл пишутся строки, которые обеспечат трансляцию и последующую сборку разрабатываемой программы;

3) make-файл запускаем, тем самым, транслируем и собираем разрабатываемую программу; загрузочный файл программы должен появиться в этом же каталоге myproga;

4) запускаем программу, смотрим результаты нашей плодотворной работы;

5) если нужные результаты не наблюдаются, то возвращаемся на пункт 1, исправляем, что испортили, и продолжаем работу дальше.

В итоге мы должны получить некоторую (системную) программу (утилиту), с помощью которой сможем в дальнейшем выполнять нужную нам обработку данных. Если мы хотим, чтобы наши усилия не канули в лету, а начали распространяться в составе некоторого дистрибутива Linux, то необходимо дополнительно сделать следующее:

- написать документацию по созданной программе — как минимум, man, а если программа достаточно сложная, то, кроме того, более подробную эксплуатационную документацию;

- определить зависимости нашей программы, то есть, выявить те библиотеки или программы, которые должны быть предварительно установлены в системе, для того, чтобы наша программа заработала правильно;

- создать инсталлер нашей программы, который будет обеспечивать проверку зависимостей, если нужно, то инициировать установку отсутствующего, производить установку программы, конфигурационных файлов, документации в нужные каталоги системы;

- оформить нашу разработку в форме пакета нужного формата (.rpm, .deb, .tgz, .pet или какого иного, используемого в нужном нам дистрибутиве): причём, пакеты должны быть двух видов: tarboll — пакет только с исполняемым файлом, без исходников, и пакет с исходными текстами программы;

- присоединиться к группе разработчиков соответствующего дистрибутива и положить созданный шедевр в репозитарий этого дистрибутива.

Естественно, предполагается, что наша разработка будет относиться к OpenSource и распространяться по какой-либо свободной лицензии.

В последнее десятилетие в связи с распространением графического режима работы появилась необходимость создания графического системного ПО, но для Unix/Linux оно почти всегда создаётся в форме «оболочек», то есть, в форме некоторой графической программы, интегрирующей работу одной или нескольких консольных программ (примеры: k3b, многие видеоплееры, gparted и др.). Разработка такой программы-оболочки не имеет особой специфики и практически ничем не отличается от разработки обычного прикладного ПО.

1.9.3. Транслятор языка С

Язык программирования С был создан Д.Ритчи в 1972 г. специально для написания операционной системы Unix, и с тех пор и "каноническая" ОС Unix, и все ее клоны пишутся на языке С. Поэтому во всех версиях Unix и Unix-подобных систем транслятор языка С входит в комплект поставки системы.

Одним из первых программных продуктов, созданных в рамках проекта GNU, также явился транслятор языка С с открытым кодом. Этот транслятор включается в поставку всех версий ОС Linux.

Транслятор языка С выполняет как собственно трансляцию - перевод исходного текста на машинный язык, - результатом чего является объектный модуль, так и редактирование связей - сборку из нескольких объектных модулей (в том числе, и библиотечных) исполняемого модуля.

Файлы с исходными текстами программ на языке С должны иметь расширение .c, например: hello.c. Результатом трансляции является файл, содержащий объектный модуль, его имя совпадает с именем исходного модуля, а расширение - .o, например: hello.o. Для файла, содержащего исполняемый модуль, стандартного расширения не существует. При трансляции программы, состоящей из единственного исходного модуля, объектный модуль автоматически удаляется после создания транслятором исполняемого модуля.

Общий формат команды вызова транслятора имеет следующий вид:

gcc [опции] [выходной_файл] файл1 [файл2 :]

У транслятора gcc очень много опций и наиболее часто употребляемые опции следующие:

-c

Подавляет фазу редактирования связей, создает объектный модуль для каждого исходного модуля из перечисленных в параметрах вызова. выходной_файл с этой опцией не задается. Опция может применяться вместе с опцией -I

-o

Компиляция и редактирование связей. Создает объектный модуль для каждого исходного модуля из перечисленных в параметрах вызова и имеющих расширение .c. Файлы .c рассматриваются как исходные модули и компилируются; файлы, имеющие расширение .o, рассматриваются как объектные модули и подключаются при редактировании связей. Параметр выходной_файл задает имя файла исполняемого модуля. Опция может применяться вместе с опциями -L, -l, -I.

-Lкаталог

Добавить каталог в список каталогов, которые содержат объектные библиотечные модули.

-lбиблиотека

При трансляции подключить модули из библиотеки.

-lбиблиотека

При редактировании связей подключить модули из библиотеки.

-Iкаталог

Искать файлы-хидеры (#include), имена которых не начинаются с /, сначала в указанном каталоге, а лишь затем - в стандартных каталогах для файлов-хидеров.

-E

Выполнить обработку указанных исходных модулей только препроцессором, результат направляется в стандартный вывод. Выходной_файл с этой опцией не задается. Опция может применяться вместе с опцией -I.

-w

Подавить выдачу предупреждающих сообщений.

Примеры:

gcc hello.c

Компиляция исходного модуля hello.c с выдачей сообщений об ошибках на стандартный вывод. Файл объектного модуля не создается. Исполняемый файл — hello.

gcc -c hello.c

Компиляция исходного модуля hello.c с выдачей сообщений об ошибках на стандартный вывод. При успешной компиляции объектный модуль записывается в файл hello.o.

gcc -o hello hello.o

Редактирование связей для объектного модуля hello.o, исполняемый модуль записывается в файл hello.

gcc -o hello hello.o hello1.c

Создание исполняемого модуля в файле hello из объектного модуля hello.o и модуля hello1.c (последний модуль является исходным, он предварительно компилируется).

gcc -o hello hello.o hello1.o -l hellolib

Создание исполняемого модуля в файле hello из объектных модулей hello.o и hello1.o с подключением объектных модулей из библиотеки hellolib.

---

2. ОБЩИЕ ТРЕБОВАНИЯ К СДАЧЕ И ОФОРМЛЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

2.1.1. Лабораторные работы предназначены для получения практических навыков работы с операционной системой.

К выполняемым лабораторным работам предъявляются следующие требования:

1) Работа выполняется самостоятельно и индивидуально по выбранной теме в полном объеме.

2) Не разрешается выполнение одного и того же задания по одной теме более чем одному человеку.

3) По каждой лабораторной работе оформляется и сдается отчет преподавателю.

4) Работа выполняется самостоятельно в произвольное время и сдается в строго оговоренные сроки только в лаборатории в часы занятий.

5) Выполнение лабораторной работы предполагает достаточно подробное изучение и правдоподобное отражение предметной области.

6) Для проверки полноты усвоения материала и самостоятельности выполнения работы преподаватель может задать дополнительные вопросы и предложить выполнить дополнительные задания.

7) Лабораторные работы выполняются в операционной среде, используемой в лаборатории «Аппаратные средства информационных систем» кафедры ТТС. Допускается использование других операционных сред, но в этом случае студентом самостоятельно должны решаться проблемы совместимости.

2.1.2. Отчет должен содержать следующие разделы:

- титульный лист;

- введение: формулировка темы (то есть, формулировка своего варианта разрабатываемой ИС);

- основная часть отчета (содержание этой части поясняется отдельно для каждой лабораторной работы).

Параметры страниц A4, поля 30-10-20-20 мм, междустрочный интервал «точно»=0,60см, шрифт DejaVu Serif или Liberation Serif 12 пунктов.

2.1.3. Отчет сдается в электронном и бумажном виде. По мере сдачи лабораторных работ все отчеты выкладываются на личном сайте в Интернете.

3. ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторная работа № 1

Тема: СОЗДАНИЕ ПОЛЬЗОВАТЕЛЯ

Цель: Научиться создавать учетные записи пользователей

Задание:

Для создания учетных записей пользователей выполните следующие действия.

1. Создать пользователя с использованием графической оболочки.

1.1. Использовать команды меню «Центр управления системой» → «Пользователи» → «Локальные учётные записи».

1.2. Прочесть справку.

1.3. Логин определить следующим образом:

Первая буква (**малая латинская буква**):

№ п/п	Название группы студентов	Первая буква логина
1	МОАИС-11	z
2	ПМ-11	y
3	ПМ-12	x
4	ИС-21	w
5	ИТСС-21	v
6	КБ-21	k
7	ИТСС-31	u
8	МОАИС-31	t
9	КБ-31	s
10	ИТСС-41	r
11	ТК-61	q
12	ПРИ-11у	p
13	ПРИ-12у	o
14	ИС-21 (ИДО)	n
15		m

16		l

Вторая, третья и четвёртая буквы — фио (**малые латинские буквы**).

1.4. Ввести новый логин в поле «Новая учётная запись». Нажать клавишу «Создать». Логин появится в списке слева (в окне учётных записей).

1.5. Выделить Bar'ом созданную учётную запись в списке (в окне слева). Поставить галочку «входит в группу администраторов». Пароль придумать самостоятельно, записать и сохранить. Ввести пароль в поля ввода пароля (дважды). Нажать клавишу «Применить».

2. В дальнейшем работать только под своей учетной записью.

Порядок сдачи лабораторной работы

1) Создать пользователя.
2) Продемонстрировать вход/выход с созданным логином.
3) Создать ссылку на программу «Терминал (Консоль)» на рабочем столе.
4) Запустить программу Терминал.
5) Продемонстрировать возможность выполнения команды su в Терминале.
6) Скопировать на рабочий стол каталог с заданиями на лабораторные работы.
7) Открыть файловым менеджером каталог с заданиями, найти задание на эту лабораторную.
8) Открыть задание на Лабораторную и громко прочесть пункт 2 задания.
9) В Отчёте должно быть:
    а) задание на работу;
    б) описание процесса создания учетной записи пользователя в графическом режиме.

Дополнительная справочная информация

"Учетная карточка" (user account)

Linux, как Unix-подобная ОС, изначально создавалась как система многопользовательская. Это значит, что в операционной системе:

- разграничиваются права различных пользователей, то есть, какие файлы они могут читать/писать/изменять, какие программы запускать и т.п.

- даётся возможность каждому пользователю создавать свою "среду обитания" (environment), то есть, иметь свои настройки для своих программ, свои папки-директории, свои настройки терминала и т.п.

Для этого в системе должна быть некоторая база данных (пусть даже очень примитивная), в которой хранятся основные сведения о каждом пользователе, допущеном к работе на данной

машине.

Одна запись в этой БД, содержащая сведения о пользователе, называется user account. Для того, чтобы допустить нового пользователя в систему, необходимо создать для него этот account. Слово "account" обычно переводится как "банковский счет" - не очень подходящий перевод. Поэтому, более подходяшим по смыслу будет - "личная учетная карточка пользователя".

При создании учётной карточки (при регистрации пользователя, точнее, при регистрации ЛОГИНа пользователя) каждому пользователю присваивается уникальный номер (user ID, uid), именно этот цифровой номер и используется внутри операционной системы при проверке прав доступа, сборе и хранении статистики и т.д. Однако, для человека оперировать номерами пользователей неудобно. Поэтому, при регистрации пользователя в системе регистрируется алфавитно-цифровое имя пользователя — логин (login, username), которое тоже должно быть уникальным. Это символьное имя может быть даже осмысленным и потому человеку более понятно.

Если какой-нибудь программе требуется в качестве аргумента указать пользователя, обычно используется это "login name". И, наоборот, если какая-либо программа в своем выводе как-нибудь упоминает конкретных пользователей, она обычно называет их этим именем, а не просто печатает номер юзера.

То есть, между user ID и login всегда существует однозначное соответствие. Просто user ID используется во внутренних данных системы, а login при общении человек-компьютер.

Структура учетной карточки пользователя (основные атрибуты):

Name:Password:user ID:group ID:General information:Home directory:Shell

Назначение полей учётной карточки

Name или login. Уникальное имя пользователя. Его система спрашивает при входе пользователя и оно же используется в командах администрирования.

Password. Пароль. Для того, чтобы никто другой, кроме этого пользователя не мог войти в систему под логином этого пользователя, естественно, у каждого пользователя должен быть его собственный секретный пароль для входа. Вообще-то, пароль может и отсутствовать, но это очень не рекомендуется. Особенно, если машина доступна по сети. Пароли хранятся в БД в закодированном виде, поэтому, даже администратор (root) не может его узнать. (Однако, администратору он и не нужен, у него и так права не ограничены. А если юзер забудет свой пароль, то проще попросить администратора записать новый пароль, чем пытаться извлечь старый).

User ID. Уникальный номер юзера, хранится в виде unsigned int, то есть, беззнакового целого в диапазоне от 0 до 65535. Он однозначно соответствует имени (login) и используется

внутри системы.

Group ID. Номер группы, к которой принадлежит пользователь. В Unix (и в Linux) все пользователи объединяются в группы. Причем,

- каждый пользователь входит по крайней мере в одну группу, но
- может быть "членом" нескольких различных групп.

Каждая группа имеет свое имя (group name) и числовой номер (groupID), которые однозначно соответствуют друг другу. Группы используются при определении прав доступа пользователей к различным объектам в системе.

General information. Это некоторые "анкетные данные" того реального человека или иногда реальной организации, которые скрываются под Name. Очень часто это поле не заполняется вообще. Иногда там пишут реальное имя/фамилию юзера (например Barack Obama) в помощь администратору (должен же админ знать, что за пользователи работают в его системе?).

Полностью это поле может состоять из

Full Name - Имя Фамилия,

Location - адрес (имеется в виду рабочее место),

Office Phone - рабочий телефон,

Home Phone - домашний телефон.

А если пользователь — юридическое лицо, то название, телефон и, иногда, адрес фирмы.

Home dir. Домашний каталог пользователя. Именно в этом каталоге помещаются настроечные файлы для различных программ с настройками под конкретного пользователя (профиль пользователя). Здесь же пользователю дается полная свобода создавать/удалять свои файлы.

Shell. Программа, которая запускается для пользователя, когда он входит в систему. Обычно это "исполнитель комманд" (shell, программа-оболочка операционной системы), который принимает команды с терминала и запускает программы для исполнения этих команд.

В современных Unix'ах есть несколько таких оболочек (sh, csh, tcsh, bash …), из которых можно выбрать наиболее подходящий. Однако, вместо shell'а здесь может быть указана любая другая программа, что часто используется для некоторых задач.

Место хранения учётных карточек

Доступная часть учётной информации хранится в файле /etc/passwd. Недоступная обычным пользователям часть учётной информации хранится в других местах, причём, в разных дистрибутивах Unix и Linux — в разных местах.

Подробнее обо всем этом можно прочитать в соответствующем man'уале (man 5 passwd).

Лабораторная работа № 2

Тема: ТЕРМИНАЛ: КОМАНДЫ РАБОТЫ С ФАЙЛАМИ

Цель: Научиться работать в терминале с командами работы с файлами ОС Linux

Задание:

Для освоения команд работы с файлами выполните следующие действия.

Итак, предполагается, что **вы работаете в текстовом (терминальном) режиме. Если вы ещё в графическом (в KDE, Gnome или где-то ещё), то перейдите в текстовый режим. Для этого нажмите одновременно клавиши Ctrl-Alt-F2**

1. В левом верхнем углу терминала отображается приглашение вида

```
<имя_компа>  login:
```

2. Входим в систему: вводим логин и пароль.

Появляется приглашение вида:

```
[login@имя_компа ~]$
```

Это означает, что мы в системе. Мы вошли под именем <login> на компьютер <имя_компа>. «Собачка» @ между login и именем компьютера является напоминанием о том, что вообще-то эта комбинация является локальным почтовым адресом пользователя в данной вычислительной системе. Значок ~ (тильда) означает, что мы находимся в своём домашнем каталоге, который называется /home/<login>.

Если мы находимся в каком-либо другом каталоге, то вместо тильды будет стоять название этого каталога. Для нас система запустила оболочку bash. Значок $ - приглашение оболочки вводить команды. Курсор в виде мигающего белого прямоугольника установится после знака «$». То есть, ваши команды будут вводиться после этого знака.

Знак «$» - это приглашение к вводу для обычного пользователя (в отличие от root'a, для которого знак приглашения - «#»).

Таким образом, после знака «$» вы вводите команду, Linux её выполняет, что-то выдаёт на экран (если у неё есть вам сказать пару слов, так она вам скажет, а если нет, то выполнит вашу команду молча) и, если команда выполнена, то Linux снова выдаёт знак «$», приглашая вас вводить следующую команду.

Ввод команды всегда завершается нажатием клавиши Enter, только после Enter система начинает выполнять команду. Если вы Enter нажали, курсор перешёл на следующую строку

и . . . всё,

знак $ не появляется. Это означает, что Linux ждёт ваших дальнейших действий — **команда ещё не завершена.** Примеры подобных команд есть в лабораторных работах.

Синтаксис команды:

<имя_команды><пробел><ключи_команды><пробел><аргументы_команды>

где

<имя_команды> - название встроенной команды программы-оболочки или название какой-либо

программы в системе;

<пробел> - символ разделитель;

<ключи_команды> - один или несколько ключей команды, вводятся в виде «-буква» (минус — признак ключа, буква — название (идентификатор) ключа); если вводятся несколько ключей подряд, то они разделяются пробелом; некоторые ключи могут иметь аргументы;

<аргументы_команды> - некоторое алфавитно-цифровое имя, идентифицирующее какой-либо об'ект в системе.

Пример команды:

ls -i -l -a /home

Далее в задании будет опускаться содержимое квадратных скобок ([login@имя компа каталог]), а будет указываться только символ «$».

Внимание: При сдаче работы возможно придётся отвечать на вопросы о назначении и смысле команд.

Справка по команде:

$ man <команда> <Enter>

В том числе можно получить справку и по самой системе man:

$ man man <Enter>

3. Ввести команду:

$ script

Скрипт запущен, файл - typescript

$

4. Прежде всего, убедитесь, что вы находитесь в своём домашнем каталоге. Это можно сделать командой:

$ pwd

5. Чтобы посмотреть, какие файлы уже находятся в вашем домашнем каталоге, нужно ввести команду:

$ ls

6. Ввести команду:

$ ls -i -l

7. Обратить внимание на первую колонку вывода этой команды. В отчёте объяснить, что означают эти числа.

8. Ввести команду:

$ ls -i -l -a

9. Обратить внимание на появившиеся каталоги и файлы с точкой. В отчёте объяснить, что это за файлы и каталоги.

10. Наиболее часто используемая форма команды ls:

$ ls -l

11. Создать файл с именем = фио (например, obama_b.txt) командой

$ touch <имя_файла>

12. Ввести в этот файл следующую информацию «Я, <фамилия имя отчество>, студент УлГУ, ФМиИТ, группа <группа>» командой:

$ cat > <имя_файла>

<информация, указанная выше>

<Ctrl-D>

13. Проверить, что информация в файл введена, командой:

$ cat <имя_файла>

14. Дополнить созданный файл следующей информацией «Это результат выполнения лабораторной № 2» с помощью команды:

$ echo <информация> >> <имя_файла>

Внимание: Если <информация> содержит пробелы (пробел — это разделитель!), то <информацию> заключить в кавычки; тогда всё, что содержится в кавычках будет восприниматься командой echo как один аргумент.

15. Проверить, что информация в файл введена правильно, командой:

$ cat <имя_файла>

16. Дополнить созданный файл следующей информацией «Дисциплина «Операционные системы», курс N-ый, семестр K-ый» с помощью команды:

$ tee >> <имя_файла>

<информация, указанная выше>

<Ctrl-C>

17. Проверить, что информация в файл введена правильно, командой:

$ cat <имя_файла>

18. И в конце файла поставить дату и время:

$ date >> <имя_файла>

19. Нажать на клавиатуре клавишу PrintScreen. В открывшемся окне программы Ksnapshot выбрать режим скринирования «Окно под курсором», нажать клавишу «Новый снимок», указать мышкой окно терминала, после восстановления окна программы Ksnapshot клавишей <Сохранить как . . .> сохранить скрин экрана в файл labaN.jpg в свой домашний каталог.

20. Завершение задания: ввести Ctrl-D. В терминале появится сообщение

«Скрипт выполнен, файл - typescript»

21. Таким образом в вашем домашнем каталоге образовался файл с именем typescript. Это протокол вашей работы. А также был создан скрин экрана перед завершением работы. На скрине **должны хорошо читаться** последние ваши команды в терминале.

Порядок сдачи лабораторной.

В отчёте должно быть:

а) **задание на работу;**

б) распечатка файла typescript;

в) распечатка **скрина окна терминала** с качеством, достаточным, чтобы можно было прочесть информацию в окне терминала.

г) объяснение (комментарии) проделанной работы.

По требованию преподавателя повторить работу в лаборатории.

Дополнительная справочная информация

Краткое описание основных команд

для работы с файлами

Здесь приведены не все команды, более полное описание смотрите в справочных руководствах (man <команда>).

cd [каталог]

Меняет текущий каталог на указанный. Если параметр опущен, то текущим становится домашний каталог.

ls [-alFR] [файл ...]

Выводит список файлов в указанном (или текущем) каталоге. Ключ -a заставляет выводить все файлы (в том числе, скрытые), ключ -l служит для вывода подробной информации о файлах, ключ -F приводит к тому, что к именам каталогов добавляется символ ’/’, к именам ссылок ’@’, к именам выполняемых файлов ’*’. При использовании ключа -R выводится список файлов не только указанного каталога, но и его подкаталогов.

touch файл ...

Меняет время доступа и изменения файла. **Если файл не существовал, то он будет создан.**

mkdir каталог

Создает каталог.

rmdir каталог

Удаляет каталог.

cp [-Rp] файл1 файл2

cp [-Rp] файл ... каталог

Копирует один файл в другой (затирая прежнее содержание этого другого файла) или копирует файлы в указанный каталог. Ключ -R предназначен для копирования каталогов, ключ -p позволяет сохранять владельцев файлов, режим доступа и время доступа и изменения.

rm [-r] файл ...

Удаляет файлы. Ключ -r позволяет удалять каталоги.

mv файл1 файл2

mv file ... directory

Переименовывает файл или перемещает файлы в заданный каталог.

cat [файл ...]

Объединяет содержимое указанных файлов и выводит на стандартный вывод.

cat [файл1+файл2+…+файлN файлM]

Объединяет содержимое указанных файлов и выводит их в один файл (объединяет).

Лабораторная работа № 3

Тема: ТЕРМИНАЛ: ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Цель: Научиться работать в терминале с командами работы с профилем пользователя ОС Linux

Задание:

Для освоения команд работы с профилем пользователя ОС Linux выполните следующие действия.

Итак, предполагается, что **вы работаете в текстовом (терминальном) режиме. Если вы ещё в графическом (в KDE, Gnome или где-то ещё), то перейдите в текстовый режим. Для этого нажмите одновременно клавиши Ctrl-Alt-F2**

1. В левом верхнем углу терминала видите приглашение вида

```
<имя_компа>  login:
```

2. Входим в систему: вводим логин и пароль.

Появляется приглашение вида:

```
[login@имя_компа ~]$
```

Прочтите пункт 2 задания на лабораторную 2.

Внимание: При сдаче работы возможно придётся отвечать на вопросы о назначении и смысле команд. Справка по команде:

$ man <команда> <Enter>

В том числе можно получить справку и по самой системе man:

$ man man <Enter>

3. Прежде всего, убедитесь, что вы находитесь в своём домашнем каталоге. Это можно сделать командой:

$ pwd

Если вы не в домашнем каталоге, то перейти в домащний каталог.

4. Прочитать в manual'е описание команды *touch*. Создать файл с именем = фио (например, ivanov_i.txt) командой

$ touch <имя_файла>

5. Вывести в этот файл вывод команды pwd.

6. Добавить в этот файл следующую информацию «Я, <фамилия имя отчество>, группа <группа>, лабораторная №7».

7. Прочитать в manual'е описание команды *date*. Добавить в этот файл дату командой «date».

8. Добавить в этот файл две пустых строки.

9. Добавить в этот файл вывод команды «ls -la».

10. Добавить в этот файл две пустых строки, а затем строку «ПРОФИЛЬ ПОЛЬЗОВАТЕЛЯ <login>:», где login — логин пользователя, под которым вы вошли в систему.

11. Прочитать в manual'е описание команды *set.* При помощи команды set добавить в этот файл профиль текущего пользователя (переменные среды).

12. Добавить в этот файл две пустых строки.

13. Прочитать в manual'е описание команды *uname.* Добавить в этот файл вывод комады «uname -a».

14. Добавить в этот файл две пустых строки.

15. Прочитать в manual'е описание команды *free.* Добавить в этот файл вывод комады «free».

16. Добавить в этот файл две пустых строки.

17. Прочитать в manual'е описание команды *df.* Добавить в этот файл вывод комады «df».

Порядок сдачи лабораторной.

В отчёте должно быть:

а) задание на работу;

б) распечатка созданного файла с именем = fio;

в) объяснение (комментарии) проделанной работы.

По требованию преподавателя повторить работу в лаборатории и объяснить, что, собственно, делал.

Дополнительная справочная информация

Общие сведения

Unix — многопользовательская, многозадачная операционная система с разделением времени. В любой момент в системе выполняется множество процессов, каждый процесс принадлежит некоторому пользователю. **Пользователь** - это объект обладающий определенными правами в системе. Каждый пользователь идентифицируется уникальным идентификатором пользователя (UID — user identifier). Пользователю присваиваются имя и пароль. Пользователь с UID 0 (root) обладает неограниченными правами. Кроме того каждый пользователь входит в одну или несколько групп. Принадлежность к группе добавляет пользователю определенные права в системе. Каждая группа идентифицируется уникальным идентификатором группы (GID — group identifier).

Информация о пользователях хранится в файле /etc/passwd. Каждая строка файла содержит информацию об одном пользователе: регистрационное имя (логин), зашифрованный пароль (в настоящее время это поле не используется, пароль хранится в другом месте), UID, GID, полное

имя пользователя, домашний каталог, командная оболочка.

Командная оболочка (командный интерпретатор, shell) — средство интерактивного взаимодействия с системой.

Домашний каталог — каталог в котором хранятся файлы пользователя. Имя домашнего каталога обычно совпадает с логином. При входе пользователя в систему этот каталог становится текущим для оболочки.

Файловая система

Файловая система — это набор структур данных на носителе + алгоритмы в ОС (драйверы файловой системы), с помощью которых ядро операционной системы организует и представляет пользователям ресурсы внешней памяти системы. Сюда относится память (внешняя, в отличие от «внутренней» - ОЗУ) на различного рода носителях информации. Емкость и количество носителей различно в разных системах. Ядро объединяет эти ресурсы в единую иерархическую структуру, которая начинается в каталоге **/ (слэш — корень файловой системы)** и разветвляется, охватывая произвольное число подкаталогов.

Структуры некоторых файловых систем рассмотрены в лекциях.

Цепочка имен каталогов, через которые необходимо пройти для доступа к заданному файлу, вместе с именем этого файла называется путевым именем файла (pathname). Путевые имена могут быть полными (абсолютными, каноническими) или относительными. Полное имя легко идентифицировать: оно всегда начинается с символа "/".

В любой момент каждый процесс привязан к некоторому текущему каталогу. Относительные имена интерпретируются с текущего каталога.

Пример полного имени:

/home/student/andreevas/aas.txt

Однако, если пользователь student находится в своём домашнем каталоге student (/home/student), то для того, чтобы поиметь доступ к файлу aas.txt не обязательно указывать полное имя, достаточно ввести относительное имя файла aas.txt:

andreevas/aas.txt

Пример: В каталоге andreevas находятся файлы aas.txt и hosts1. Команда pwd показывает, где вы находитесь:

```
$ pwd
/home/student/andreevas/
```

Вам нужно добавить содержание файла hosts1 в файл aas.txt. Это делается командой:

```
cat hosts1 » aas.txt
```

Относительное имя файла может быть очень коротким, например, просто f. Если в имени файла

вообще нет знака "/" (слэш), то имя относится к файлу текущего каталога. Если слэш есть (но не в начале имени – как в вышеуказанном примере), то все, что находится слева от первого в имени слэша, расценивается как подкаталог текущего каталога.

Особенности:

а) символ ~ (тильда), как правило, обозначает домашний каталог пользователя; например, если пользователь находится где-то в файловой системе и ему нужно вернуться в свой домашний каталог, то это можно сделать командой

cd ~

б) каталог . (точка) — текущий каталог; каталог .. (две точки) — вышестоящий каталог.

в) предположим, пользователь student, **находясь в своём домашнем каталоге**, транслировал программу proga и она создалась в его домашнем каталоге (то есть, полный путь к ней /home/student/proga); тогда, чтобы запустить программу proga на выполнение нужно дать команду

./proga <Enter>

то есть, символы **./** указывают, что путь к программе должен исчисляться от текущего каталога (того каталога, где сейчас находится пользователь).

Иерархическая структура файловой системы может быть произвольного размера. Однако существуют определенные ограничения, зависящие от конкретной операционной системы. Как правило имена файлов и каталогов не должны содержать более 256 символов, а в определении одного (полного) пути не должно быть более 1023 символов. Имена файлов могут состоять из любых символов, за исключением слэша и символа с кодом ноль. Однако на самом деле **в именах файлов НАСТОЯТЕЛЬНО РЕКОМЕНДУЕТСЯ использовать только следующие символы и ничего кроме них: a-z,A-Z,0-9 и спецсимволы «-», «_» и «.»**, в противном случае возникают проблемы.

Максимальная длина имени файла определяется конкретной системой. Для каждого файла определен владелец этого файла и группа владельца данного файла. Для каждого файла определяются права доступа владельца файла, права доступа группы, права доступа всех остальных. Есть три типа прав доступа: чтение, запись, выполнение/поиск. Изменить права доступа к файлу может только владелец и суперпользователь (root) командой с**hmod**. Изменить владельца файла и группу владельца может только пользователь root командой **chown.**

В ОС Unix существует семь типов файлов:

1) Обычный файл (обозначаются **-**) — это просто последовательность байтов. Обычный файл может содержать выполняемую программу, главу книги, графическое изображение и т.п.

2) Каталоги (обозначаются **d**) могут содержать файлы любых типов в любых сочетаниях.

Специальные имена . и .. обозначают соответственно сам каталог и его родительский каталог.

3,4) Файлы устройств символьные и блочные (символьные обозначаются **c**, блочные обозначаются **b**) - позволяют программам взаимодействовать с аппаратными средствами и периферийными устройствами системы. Файлы устройств находятся в каталоге /dev и нигде больше. При конфигурировании ядра к нему добавляются те модули, которые знают, как взаимодействовать с каждым из устройств системы. За всю работу по управлению конкретным устройством отвечает специальная программа, называемая драйвером устройства. Драйверы устройств образуют стандартный коммуникационный интерфейс, который выглядит как обычный файл. Когда ядро получает запрос к байт-ориентированному или блок-ориентированному файлу устройства, оно просто передает этот запрос соответствующему драйверу устройства.

Каждому типу устройств системы может соответствовать несколько файлов устройств. Поэтому файлы устройств характеризуются двумя номерами: старшим и младшим. Старший определяет драйвер (тип устройства), а младший конкретное устройство (экземпляр устройства этого типа).

Файл устройства является псевдофайлом, он не размещен на диске, о нем есть только запись (строка в индексной таблице), которая используется при доступе к устройству. Первое число, которое стоит в поле длины файла в выводе программы ls для файлов устройств – это major номер, а второе, после запятой – minor номер. Первый из них означает номер типа устройств и одновременно – позицию в ядре (номер строки в таблице драйверов), в которой следует искать адрес драйвера устройства. Второй – номер экземпляра устройства данного типа. Поэтому файлы однотипных устройств в выводе команды ls имеют одинаковые major номера. Устройство каждого типа имеет свой major номер. Major номера назначаются соответствующей международной организацией. Minor номер определяется ОС в процессе загрузки при обнаружении устройства.

5) Локальные сокеты (sokets) — это файлы специального типа, которые связаны с определёнными структурами в ядре, с помощью которых обеспечивается взаимодействие между процессами в пределах одной ОС. Обращение к ним осуществляется через объект файловой системы (файл типа socket), а не через сетевой порт.

6) Именованные каналы (именованные pipe), также как и сокеты обеспечивают взаимодействие двух процессов, выполняющихся на одной машине (точнее, в пределах одной ОС).

7) Символические ссылки — это запись в каталоге, ссылающаяся на файл с определенным именем. Символическая ссылка содержит путевое имя файла, на который она ссылается, обеспечивая возможность указывать вместо путевого имени файла имя ссылки. То есть, символическая ссылка – это отдельный файл типа "символическая ссылка", и индексный дескриптор этого файла содержит только путь к файлу или каталогу, на который указывает ссылка.

Можно создать символическую ссылку на любой каталог, а также на файл, находящийся в другом разделе Unix. При удалении символической ссылки с файлом или с каталогом, на который

она ссылается, ничего не происходит. При удалении файла, на который ссылается символическая ссылка, она "повисает в воздухе", ссылаясь на пустоту. В этом случае при обращении к такой "пустой" ссылке возникнет ошибка file not found, несмотря на то, что сама ссылка будет видна и доступна в списке файлов.

Перенаправление ввода и вывода

Если некоторый процесс намерен производить ввод или вывод информации в файл, то он должен сначала открыть этот файл. При открытии файла процесс получает дескриптор файла — некоторое число (беззнаковое целое), которое используется, в дальнейшем для обращения к файлу. При запуске процесса ему операционной системой передаются дескрипторы трех открытых файлов: 0 – стандартный ввод (stdin), 1 – стандартный вывод (stdout), 2 – стандартный вывод ошибок (stderr). Как правило все эти дескрипторы указывают на терминал – tty. Оболочка позволяет назначать другие файлы для ввода и вывода при помощи команд перенаправления:

программа < файл

Таким образом, при запуске команды дескриптор 0 будет связан с файлом, т.е. программа будет считывать данные не с клавиатуры, а из файла. Файл будет открыт для чтения.

программа > файл

Здесь при запуске команды дескриптор 1 будет связан с файлом, т.е. программа будет выводить результаты работы не на экран, а в заданный файл. Файл будет открыт для записи, **(!внимание)** если файл существовал, он будет очищен, если нет, то он будет создан.

программа >> файл

При запуске команды дескриптор 1 будет связан с указанным файлом, как и в предыдущем случае. Однако в данном случае, если файл существовал, то он **не будет перезаписан**, данные будут добавляться в конец файла.

программа n > файл

При запуске команды дескриптор с номером n будет связан с указанным файлом. Например, если указать 2 > err.log, то вывод сообщений об ошибках будет производиться в файл err.log. Аналогично, можно указывать дескриптор перед операторами перенаправления > и >>.

программа n < > файл

При запуске команды дескриптор с номером n будет связан с указанным файлом. Файл будет открыт для чтения и записи.

При перенаправлении можно вместо имени файла указывать дескриптор, для этого следует поставить перед дескриптором знак &. Например: 2 > &1 скопирует содержимое дескриптора 2 в дескриптор 1. Копируемый дескриптор должен быть открыт для чтения или записи в зависимости от операции.

Операции перенаправления выполняются слева направо. В случаях, когда используется копирование дескрипторов, порядок выполнения операций может влиять на результат.

Конвейер

Конвейер — последовательность из одной или более команд, разделенных символом | (вертикальная черта - конвейер). Формат конвейера следующий:

[time [-p]] [!] command [| command2 ...]

Стандартный вывод command подключается к стандартному вводу команды command2. Это подключение производится до выполнения любых перенаправлений.

Если конвейеру предшествует зарезервированное слово ! (восклицательный знак), то код завершения конвейера равен логическому отрицанию кода завершения последней команды. Иначе код завершения конвейера равен коду завершения последней команды. Интерпретатор ожидает завершения всех команд до того как вернет значение кода завершения.

Если конвейеру предшествует зарезервированное слово **time**, то после завершения выполнения конвейера будет выведена информация о времени выполнения конвейера и о затраченном времени процессора в режимах пользователя и системы.

Каждая команда в конвейере выполняется как отдельный процесс (т.е. в подоболочке).

Переменные окружения и команды

У каждого процесса имеется область памяти называемая программным окружением (program environment) — это набор строк, заканчивающийся нулевым символом. Эти строки называются переменными окружения. Каждая строка имеет вид: имя переменной = значение. Имя переменной может состоять из алфавитно-цифровых символов и знака подчеркивания. Цифра не может быть первым символом имени. Присвоение значения переменной в оболочке производится следующим образом:

Имя = Значение

Для того, чтобы значение переменной передавалось процессам, порождаемым оболочкой, следует использовать встроенную команду **export**. Следующие две команды помечают переменные VAR и TST как экспортируемые и присваивают переменной TST значение /usr/doc:

```
export VAR
export TST=/usr/doc
```

Для того, чтобы просмотреть значения переменных окружения можно использовать команду **set**, которая выводит значения всех переменных окружения.

Для того, чтобы получить значение переменной, перед ее именем указывается знак доллара. Такое выражение будет заменяться интерпретатором на значение переменной. Например, команда **echo** выводит в стандартный вывод свои аргументы, тогда следующее выражение:

echo TST=$TST

выведет на экран TST=/usr/doc (при условии, что значение переменной TST – /usr/doc).

touch - изменяет временные штампы файла, синтаксис команды:

touch [-acm][-r ref_file|-t время] [--]файл...

в версии GNU:

touch [-acfm] [-r файл] [-t decimtime] [-d time] [--time={atime,access,use,mtime,modify}] [--date=время] [--reference=файл] [--no-create] [--help] [--version] [--] файл...

Команда изменяет время последнего доступа и/или время последней модификации каждого заданного файла. Эти временные штампы устанавливаются в текущее время; или, если задана опция -r, то эти штампы устанавливаются в те же, что имеет файл ref_file; или, если задана опция -t, то эти штампы устанавливаются на заданное время. Оба штампа изменяются, если не задана ни одна из опций -a и -m или если заданы они обе. Если задана только опция -a или только -m, то изменяться будет, соответственно, только время последнего доступа или время последней модификации. Если заданный файл еще не существует, то он создается (как пустой файл с правами доступа 0666, с учетом umask), если не задана опция -c.

Опции:

-a

Изменить время последнего доступа к файлу.

-c

Не создавать файл.

-m

Изменить время последней модификации файла.

-r ref_file

Использовать соответствующий временной штамп от файла ref_file в качестве нового значения для изменяемого временного штампа (или штампов).

-t время

Использовать заданное время в качестве нового значения для изменяемого временного штампа (или штампов). Аргумент является десятичным числом вида [[ВВ]ГГ]ММДДччмм[.СС] с обозначениями (ВВ - век, ГГ - год, ММ - месяц, ДД - день, чч - часы, мм - минуты, СС - секунды). Если ВВ не задан, то год ВВГГ берется из диапазона 1969-2068. Если СС не задано, то берется 0. Секунды могут быть заданы в диапазоне 0-61, чтобы можно было указать високосную секунду. Считается, что результирующее время соответствует часовому поясу, заданному в переменной окружения TZ. Если в результате получилось время до 1 января 1970 года, то будет выдана ошибка.

uname - сообщает информацию о данном компьютере и операционной системе, синтаксис:

uname [Опции]

Опции:

-a, --all

выводит подробную информацию в следующем порядке:

-s, --kernel-name

выводит название ядра операционной системы

-n, --nodename

выводит сетевое название узла

-r, --kernel-release

выводит выпуск ядра операционной системы

-v, --kernel-version

выводит версию ядра операционной системы, включая дату выпуска

-m, --machine

выводит сведения о типе компьютера

-p, --processor

выводит тип процессора для данного компьютера

-i, --hardware-platform

выводит сведения о платформе компьютера

-o, --operating-system

выводит тип операционной системы

df - отчет об использовании файловой системы, синтаксис:

df [ОПЦИЯ]... [ФАЙЛ]...

Команда показывает количество места на диске, доступное в файловой системе, в которой содержаться указанные ФАЙЛЫ. Если ни ФАЙЛ не указан, показывается доступное место на всех смонтированных файловых системах. Размеры указаны в блоках по 1 кб по умолчанию, за исключением заданной переменной окружения POSIXLY_CORRECT (в этом случае

используются 512-байтовые блоки).

Если ФАЙЛ указан как абсолютный путь к узлу, в который смонтирована файловая система, df показывается доступное место в этой файловой системе, а не содержимое файловой системы узла устройства (которая всегда показывается как корневая файловая система). Опции:

-a, --all

включать виртуальные файловые системы

-B, --block-size=РАЗМЕР использовать блоки указанного РАЗМЕРА (в байтах)

-h, --human-readable

печатать размеры в удобочитаемом формате (напр., 1K 234M 2G)

-H, --si

то же, но использовать степени 1000, а не 1024

-i, --inodes

вывести информацию об индексных дескрипторах, а не об использовании блоков

-k аналог --block-size=1K

-l, --local

перечислить только локальные файловые системы

-P, --portability

выводить в формате POSIX

-t, --type=ТИП

перечислить только файловые системы указанного ТИПА

-T, --print-type

печатать тип файловой системы

-x, --exclude-type=ТИП

исключить файловые системы указанного ТИПА

Лабораторная работа № 4

Тема: ТЕРМИНАЛ: РЕДАКТОР Vi (Vim)

Цель: Научиться редактировать файлы с помощью редактора vim

Задание:

Для освоения редактора vi (vim) выполните следующие действия.

Итак, предполагается, что **вы работаете в текстовом (терминальном) режиме. Если вы ещё в графическом (в KDE, Gnome или где-то ещё), то перейдите в текстовый режим. Для этого нажмите одновременно клавиши Ctrl-Alt-F2**

1. В левом верхнем углу терминала видите приглашение вида

```
<имя_компа>  login:
```

2. Входим в систему, вводим логин и пароль.

Появляется приглашение вида:

```
<login>@<имя_компа>:~$
```

Прочтите пункт 2 задания на лабораторную работу № 2.

Внимание: При сдаче лабы возможно придётся отвечать на вопросы о назначении и смысле команд.

Справка по команде:

$ man <команда> <Enter>

В том числе можно получить справку и по самой системе man:

$ man man <Enter>

3. Прежде всего, убедитесь, что вы находитесь в своём домашнем каталоге. Это можно сделать командой:

$ pwd

Если вы не в домашнем каталоге, то перейти в домащний каталог.

4. Создать файл с именем = фио (например, obama_b.txt) командой

$ **vim** <имя_файла>

5. Ввести следующую информацию «Я, <фамилия имя отчество>, группа <группа>, лабораторная № 4».

6. Добавить в этот файл две пустых строки.

7. Ввести следующую информацию (выделено курсивом):

= = =

Запуск:

vim *имя_файла*

Перемещение по файлу:

клавиши hjkl,

или клавиши со стрелками вверх-вниз-вправо-влево, если они есть на вашей клавиатуре.

Редактирование:

i - начать ввод текста перед курсором,

a - начать ввод текста после курсора,

<esc> (<Ctrl><[>) - выход из режима редактирования.

Выход из редактора:

:q!<Enter> выйти без сохранения файла

:wq<Enter> сохранить файл и выйти из редактора

Выйти можно только когда редактор находится в командном режиме, то есть, прежде чем нажимать «:», нужно сначала нажать клавишу <esc>.

Если вводился русский текст, то нужно переключить раскладку клавиатуры, прежде чем вводить команды.

= = =

8. Выйти из редактора с сохранением.

9. Добавить в этот файл дату командой «date».

Порядок сдачи лабораторной.

В отчёте должно быть:

а) задание на работу;

б) распечатка созданного файла с именем = fio;

в) объяснение (комментарии) проделанной работы.

По требованию преподавателя повторить работу в лаб326 и объяснить, что, собственно, делал.

Дополнительная справочная информация

Знакомство с редактором Vim

Одним из самых старых текстовых редакторов является редактор vi. Этот редактор обладает несколько своеобразным интерфейсом и, поначалу, работа с ним вызывает у неопытного пользователя серьёзные затруднения, но тем не менее этот редактор очень популярен и многие тысячи людей используют именно его для редактирования текстов. Практически в любой Unix

совместимой системе имеется какая-либо реализация vi или vim. **Vim отличается от vi тем, что понимает локализацию, то есть, умеет работать не только с латинскими буквами, но и с буквами других алфавитов.**

Для освоения редактора vim запустите команду vimtutor и выполните упражнения содержащиеся в открывшемся файле. Если Вам не очень понятен английский язык, воспользуйтесь приведенной ниже краткой справкой по vi.

Редактор vi

Vi экранный текстовый редактор. Большая часть экрана используется для отображения редактируемого файла. Последняя строка экрана используется для ввода команд и вывода различной информации. Редактор может находиться либо в режиме редактирования, либо в режиме ввода команд. Для того, чтобы совершать какие либо действия Вы должны находиться в нужном режиме. После запуска редактор находится в командном режиме. Для перехода из режима редактирования в командный режим используется клавиша Esc. Для того, чтобы начать редактирование файла используется команда

vi имя_файла

Основные возможности в командном режиме:

• Перемещение по файлу:

h, left-arrow

переместить курсор влево на один символ

j, down-arrow

переместить курсор вниз на одну строку

k, up-arrow

переместить курсор вверх на одну строку

l, right-arrow

переместить курсор вправо на один символ

/text<cr>

найти строку text в файле и поместить курсор на ее первый символ. После этого можно использовать клавиши n и Shift-n для перемещения к следующему или предыдущему включению строки.

• Переход в режим редактирования:

i

начать ввод текста перед курсором

a

начать ввод текста после курсора

o

вставить строку после текущей и начать ввод текста в ней

O

вставить строку перед текущей и начать ввод текста в ней

Выход из режима редактирования — клавиша <esc> (<Ctrl><[>).

• Копирование, вставка и удаление:

yy y$ yw

скопировать строку, строку от позиции курсора до конца, слово.

dd d$ dw

удалить строку, строку от позиции курсора до конца, слово.

X

удалить символ

p

вставить содержимое буфера после курсора

P

вставить содержимое буфера перед курсором

u

Отменить последнюю операцию

• Сохранение и чтение файлов, выход из редактора:

:w<Enter>

сохранить файл

:w filename<Enter>

сохранить файл под указанным именем

:r filename<Enter>

вставить содержимое указанного файла

:q<Enter>

выйти из редактора

:wq<Enter>

сохранить файл и выйти из редактора

:q!<Enter>

выйти без сохранения файла

Почему важно уметь работать с редактором vi

Если вы всегда работаете только на полномерных терминалах (то есть, и дисплей большой цветной, и мышка есть, и клавиатура полная, как минимум 101 клавиша), то знать **vi** вам не обязательно. И Word'a с вас вполне достаточно. Или Writer'a.

Однако, если вам приходится работать

- с системами удалённо,

- или на всяком-разном оборудовании, которому мышка не предусмотрена, а клавиатура - урезанная всего лишь с тридцатью клавишами или даже меньше и функциональные клавиши и рядом с ней не лежали, то запустить Word не удастся. Как, впрочем, и Writer. А возможно не удастся запустить и более простые вещи, типа kate или nano.

А, вот, **vi** будет работать даже в этих условиях!

Самые минимальные знания vi, которые легко запомнить и которые могут вас выручить в кризис

Запуск:

vi имя_файла

Перемещение по файлу:

клавиши h,j,k,l,

или клавиши со стрелками вверх-вниз-вправо-влево, если они есть на вашей клавиатуре.

Редактирование:

i - начать ввод текста перед курсором,

a - начать ввод текста после курсора,

<esc> (<Ctrl><[>) - выход из режима редактирования.

Выход из редактора:

:q!<Enter> выйти без сохранения файла

:wq<Enter> сохранить файл и выйти из редактора

Выйти можно только тогда, когда редактор находится в командном режиме, то есть, прежде чем нажимать «:», нужно сначала нажать клавишу <esc>.

Если вводился русский текст, то нужно переключить раскладку клавиатуры, прежде чем вводить команды.

Лабораторная работа № 5

Тема: ТЕРМИНАЛ: АТРИБУТЫ ФАЙЛОВ

Цель: Научиться читать и изменять атрибуты файлов

Задание:

Для изучения атрибутов файлов выполните следующие действия.

Итак, предполагается, что вы **работаете в текстовом (терминальном) режиме. Если вы ещё в графическом (в KDE, Gnome или где-то ещё), то перейдите в текстовый режим. Для этого нажмите одновременно клавиши Ctrl-Alt-F2**

1. В левом верхнем углу терминала видите приглашение вида

```
<имя_компа>  login:
```

2. Входим в систему: вводим логин и пароль.

Появляется приглашение вида:

```
[login@имя_компа ~]$
```

Прочтите пункт 2 в задании на лабораторную 2.

Далее в задании будет опускаться содержимое квадратных скобок ([<login>@<имя компа>]), а будет указываться только символ «$».

Внимание: При сдаче лабы возможно придётся отвечать на вопросы о назначении и смысле команд.

Справка по команде:

$ man <команда> <Enter>

В том числе можно получить справку и по самой системе man:

$ man man <Enter>

3. Прежде всего, убедитесь, что вы находитесь в своём домашнем каталоге. Это можно сделать командой:

$ pwd

Если вы не в домашнем каталоге, то перейти в домашний каталог.

4. Перейти в режим root.

5. Выполнить команды

cp /var/log/syslog/messages messages

cp /var/log/dmesg dmesg

cp /etc/passwd passwd

6. Прочитать в manual'е описание команды *chown*. Сменить хозяина скопированных файлов на себя (на свой логин) командой chown.

7. Выйти из root.

8. Создать файл с именем = фио (например, ivanov_i.txt) командой

$ **touch** <имя_файла>

9. Ввести следующую информацию «Я, <фамилия имя отчество>, группа <группа>, лабораторная №5».

10. Добавить в этот файл две пустых строки.

11. Прочитать в manual'е описание команды *tail*. Добавить в этот файл вывод следующих команд

tail messages

tail dmesg

12. С помощью команды **cut** выделить из файла passwd первое и третье поле, вывод этой команды отсортировать по алфавиту и добавить в созданный ранее файл фио.

11. Добавить в файл фио количество строк, слов и байт, содержащихся в файлах messages, dmesg и passwd.

12. Добавить в этот файл дату командой «date».

Порядок сдачи лабораторной.

В отчёте должно быть:

а) задание на работу;

б) распечатка созданного файла с именем = fio;

в) объяснение (комментарии) проделанной работы.

По требованию преподавателя повторить работу в лаборатории и объяснить, что, собственно, делал.

Дополнительная справочная информация

Команды обработки текстовых файлов

Здесь приведены краткие описания некоторых команд (более подробно: man <команда>)

head [-n count] [file...]

Выводит первые count строк файла (по умолчанию 10).

tail [-f] [-n count] [file...]

Выводит последние count строк файла (по умолчанию 10). Если указан ключ -f, то ожидает добавления данных в конец файла и выводит их.

comm [-123] file1 file2

Считывает файлы file1 и file2, которые должны быть предварительно отсортированы, и выводит три колонки текста. В первой колонке строки имеющиеся только в file1, во второй имеющиеся только в file2, в третьей имеющиеся в обоих файлах. Параметры -1, -2, -3 позволяют подавить вывод соответствующей колонки.

cut {-c list|-f list [-d delim]} [file...]

Вырезает из каждой строки указанные символы и выводит их. Аргумент list — список чисел и диапазонов чисел разделенных запятыми. Для -c числа указывают номера символов подлежащих выводу, для -f номера полей. Поля разделены символом delim (по умолчанию символ табуляции).

sort [-c|-m] [-o output] [-urnb] [file...]

Производит сортировку строк файлов, их объединение или проверяет отсортирован файл или нет. Значения параметров:

-c только проверить правильность сортировки
-m объединить предварительно отсортированные файлы
-u удалять повторяющиеся элементы
-r сортировка в обратном порядке
-n сортировка чисел
-b игнорировать лидирующие пробелы
-o file производить вывод в файл file

wc [-c|-m][-lw][file...]

Читает один или более входных файлов и, по умолчанию, выводит число символов новой строки, слов и байт содержащихся в каждом файле на стандартный вывод. Значения параметров:

-c вывести число байт в каждом входном файле
-l вывести число символов новой строки в каждом входном файле
-m вывести число символов в каждом входном файле
-w вывести число слов в каждом входном файле

iconv -f codepage1 -t codepage2 [file...]

Конвертирует файлы из кодировки codepage1 в кодировку codepage2 и выводит результат на стандартный вывод. Например, команда

```
iconv -f windows-1251 -t koi8-r file
```

перекодирует файл из кодировки CP1251 в кодировку KOI8-R.

Лабораторная работа № 6

Тема: BASH-ПРОГРАММИРОВАНИЕ

Цель: Научиться создавать простые скрипты

Задание:

Разработать скрипт, выполняющий указанные действия согласно вариантам. Скрипт должен выдавать ФИО студента, комментарии по поводу предстоящих действий, и делать задержку после выдачи результатов на экран, иметь меню. Скрипт должен содержать функции, выполняющие указанные действия.

Вар	Действия	Вар	Действия	Вар	Действия	Вар	Действия
1	23,32,19.5,	8	7,19.3,30,	15	4.5,14.3,36	22	1,9,25.2,
2	24.1,30,19.6,	9	8.1,19.4,31,	16	4.6,15,25.1,	23	2,10,25.3,
3	24.2,29,26.1,	10	8.2,19.5,32,	17	4.7, 16,26.1,	24	3,11,26.3,
4	24.3,28,26.2,	11	9,19.6,19.1,	18	5,17,26.2,	25	4.1,12,34.1,
5	24.4,27,26.3,	12	19.7,20,19.2,	19	6.1,18,27,	26	4.2,13,34.2,
6	24.5,26.1,35.1	13	19.8,21,19.3,	20	6.2,19.1,28,	27	4.3,14.1,35.1,
7	25.1,26.2,35.2	14	22,33,19.4,	21	6.3,19.2,29,	28	4.4,14.2,35.2,

Список действий:

1. Выдать краткое описание указанной команды.
2. Отобразить тип аппаратной платформы.
3. Вычислить контрольную сумму указанного файла.
4. Вывести текущую дату в формате:
    1) дня недели,
    2) названия месяца,
    3) номера месяца,
    4) года,
    5) в национальном формате,
    6) номера дня недели,
    7) номера недели.
5. Вывести указанное сообщение.
6. Вывести информацию об указанном пользователе в формате:

1) полном,

2) только группы,

3) только номера.

7. Вывести информацию на заданную тему:

1) из указанного каталога,

2) краткой справки по указанной команде,

3) номера версии этой утилиты.

8. Вывести страницы руководства в формате:

1) все страницы,

2) краткого описания команды.

9. Отобразить список процессов.

10. Сохранять выводимые на экран символы в указанном файле путем дописывания в конец.

11. Приостановить выполнение команд на 10 сек.

12. Найти текстовую строку в заданном объектном файле.

13. Отобразить график загрузки системы.

14. Отобразить информацию :

1) о периоде времени после последней перезагрузке,

2) о числе пользователей, подключенных к системе,

3) о средней загрузке системы.

15. Выдать информацию о подключенных пользователях.

16. Вывести информацию о системе.

17. Сменить текущий каталог.

18. Отобразить режим доступа к указанному файлу.

19. Изменить режим доступа к личному файлу:

1) группе пользователей,

2) прочим пользователям,

3) установить разрешение на чтение,

4) установить разрешение на запись,

5) установить разрешение на исполнение,

6) установить запрет на чтение,

7) установить запрет на запись,

8) установить запрет на исполнение,

20. Отобразить владельца файла.

21. Сменить владельца файла.

22. Выполнить копирование указанного файла с сохранением атрибутов.

23. Выполнить копирование указанного каталога.

24. Вывести список файлов и подкаталогов указанного каталога путем:

    1) вывода списка по строкам,

    2) вывода без сортировки,

    3) вывода с сортировкой по расширениям,

    4) вывода с сортировкой по дате,

    5) вывода с пометками для файлов, каталогов и символических списков.

25. Найти во всей файловой системе:

    1) указанный файл,

    2) файлы по указанному пути,

    30 файл, изменяемый указанное количество дней назад,

26. Отобразить список файлов указанного каталога путем:

    1) вывода по столбцам,

    2) в полном формате,

    3) отсортированный по дате изменения.

27. Создать указанный каталог.

28. Переместить указанный файл.

29. Вывести имя текущего каталога.

30. Удалить указанный каталог.

31. Вывести размер указанной программы.

32. Упаковать указанный файл или каталог.

33. Распаковать указанный файл или каталог.

34. Вывести содержимое указанного файла:

    1) на экран с нумерацией строк,

    2) в конец другого файла.

35. Сравнить содержимое двух указанный файлов:

    1) с выводом позиций всех отличий,

    2) с выводом отличающихся символов.

36. Отформатировать текст в указанном файле, разбив его на несколько столбцов.

Порядок сдачи лабораторной.

1. Демонстрируется выполнение скрипта.

2. Предоставляется отчет. В отчёте должно быть:

    а) задание на работу;

    б) распечатка созданного файла с именем = fio;

    в) объяснение (комментарии) проделанной работы.

По требованию преподавателя повторить работу в лаборатории и объяснить, что, собственно,

делал.

Дополнительная справочная информация

Краткая справка по языку Bash

Bourne-again shell (GNU Bash) — это реализация Unix shell, написанная на C в 1987 году Брайаном Фоксом (Brian Fox) для GNU Project.

Синтаксис языка Bash является надмножеством синтаксиса языка Bourne shell. Подавляющее большинство скриптов для Bourne shell могут быть исполнены интерпретатором Bash без изменений, за исключением скриптов, использующих специальные переменные или встроенные команды Bourne shell.

Также синтаксис языка Bash включает идеи, заимствованные из Korn shell (ksh) и C shell (csh): редактирование командной строки, история команд, стек директорий, переменные $RANDOM и $PPID, синтаксис POSIX для подстановки команд: $(…).

Комментарии. Строки, начинающиеся с #, трактуются как комментарии.

Подстановка результатов выполнения команд. Выражения можно заключать в обратные кавычки (`). Такие выражения вычисляются в месте использования. Они могут быть, например, частью строк. Пример. Пусть параметром макрокоманды является имя файла с расширением .for. Требуется удалить одноименный файл с расширением .err.

```
name=`ena -n $1`
rm -f ${name}.err
```

Значение, полученное в результате выполнения команды

```
ena -n $1
```

присваивается переменной name. Фигурные скобки использованы для выделения аргумента операции перехода от имени к значению. Без них .err приклеилась бы к имени.

Переменные и подстановка их значений. Все переменные в языке - текстовые. Их имена должны начинаться с буквы и состоять из латинских букв, цифр и знака подчеркивания (_). Чтобы воспользоваться значением переменной, надо перед ней поставить символ $. Использование значения переменной называется подстановкой.

Различается два класса переменных: позиционные и с именем. Позиционные переменные - это аргументы командных файлов, их именами служат цифры: $0 - имя команды, $1 - первый аргумент и т.д. Значения позиционным переменным могут быть присвоены и командой set .

Пример. После вызова программы, хранящейся в файле ficofl:

```
ficofl -d / \*.for
```

значением $0 будет ficofl, $1 - -d, $2 - /, $3 - *.for, значения остальных позиционных переменных

будут пустыми строками. Заметим, что если бы символ * при вызове ficofl не был экранирован, в качестве аргументов передались бы имена всех фортранных файлов текущей директории.

Еще две переменные хранят командную строку за исключением имени команды: $@ эквивалентно $1 $2 ..., а $* - "$1 $2 ...".

Начальные значения переменным с именем могут быть установлены следующим образом:

<имя>=<значение> [<имя>=<значение>] ...

Не может быть одновременно функции и переменной с одинаковыми именами.

Следующие переменные автоматически устанавливаются:

#	количество позиционных параметров (десятичное)
-	флаги, указанные при запуске shella или командой set
?	десятичное значение, возвращенное предыдущей синхронно выполненной командой
$	номер текущего процесса
!	номер последнего асинхронного процесса
@	эквивалентно $1 $2 $3 ...
*	эквивалентно "$1 $2 $3 ..."

Напомним: чтобы получить значения этих переменных, перед ними нужно поставить знак $.

Пример: выдать номер текущего процесса:

```
echo $$
```

Управляющие конструкции

Простая команда - это последовательность слов, разделенная пробелами. Первое слово является именем команды, которая будет выполняться, а остальные будут переданы ей как аргументы. Имя команды передается ей как аргумент номер 0 (т.е. имя команды является значением $0). Значение, возвращаемое простой командой - это ее статус завершения, если она завершилась нормально, или (восьмеричное) 200+статус, если она завершилась аварийно.

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, а & - параллельно.

Операция && (||) означает, что список, следующий за ней будет выполняться лишь в том случае, если код завершения предыдущего конвейера нулевой (ненулевой).

Команда - это либо простая команда, либо одна из управляющих конструкций. Кодом завершения команды является код завершения ее последней простой команды.

Цикл ДЛЯ

```
for <переменная> [ in <набор> ]
do
<список>
done
```

Если часть in <набор> опущена, то это означает in "$@" (то есть in $1 $2 ... $n). Пример. Вывести на экран все cpp-файлы текущего проекта:

```
for f in *.cpp
do
  cat $f
done
```

Оператор выбора

```
case $<переменная> in
      <шаблон> | <шаблон>... ) <список> ;;
      . . .
esac
```

Оператор выбора выполняет <список>, соответствующий первому <шаблону>, которому удовлетворяет <переменная>. Форма шаблона та же, что и используемая для генерации имен файлов. Часть | шаблон... может отсутствовать.

Пример. Определить флаги и откомпилировать все указанные файлы.

```
                # инициализировать флаг
flag=
                # повторять для каждого аргумента
for a
do
      case $a in

                        # объединить флаги, разделив их пробелами
              -[ocSO]) flag=$flag' ' $a ;;
              -*) echo 'unknown flag $a' ;;

                        # компилировать каждый исходный файл и сбросить флаги
             *.c) cc $flag $a; flag= ;;
             *.s) as $flag $a; flag= ;;
             *.f) f77 $flag $a; flag= ;;

                        # неверный аргумент
               *) echo 'unexpected argument $a' ;;
      esac
done
```

Условный оператор

```
if <список1>
then
  <список2>
[ elif <список3>
then
  <список4> ]
. . .
[ else
  <список5> ]
fi
```

Выполняется <список1> и, если код его завершения 0, то выполняется <список2>, иначе - <список3> и, если и его код завершения 0, то выполняется <список4>. Если же это не так, то выполняется <список5>. Части elif и else могут отсутствовать.

Цикл ПОКА

```
while <список1>
do
    <список2>
done
```

До тех пор, пока код завершения последней команды <списка1> есть 0, выполняются команды <списка2>.

При замене служебного слова while на until условие выхода из цикла меняется на противоположное.

В качестве одной из команд <списка1> может быть команда true (false). По этой команде не выполняется никаких действий, а код завершения устанавливается 0 (-1). Эти команды применяются для организации бесконечных циклов. Выход из такого цикла можно осуществить лишь по команде break.

Функции

```
function <имя> ()
{
    <список>;
}
```

Определяется функция с именем <имя>. Тело функции - <список>, заключенный между { и }.

Пример вычисления факториала.

Используется рекурсивное определение факториала.

```
function factorial {
  typeset -i n=$1
  if [ $n = 0 ]; then
    echo 1
    return
  fi
  echo $(( n * $(factorial $(( n - 1 )) ) ))
```

```
}
for i in {0..16}
do
    echo "$i! = $(factorial $i)"
done
```

Используется итеративное определение факториала.

```
f=1
for (( n=1; $n<=17; $((n++)) ));
do
  echo "$((n-1))! = $f"
  f=$((f*n))
done
```

Лабораторная работа № 7

Тема: ФАЙЛОВЫЙ МЕНЕДЖЕР mc

Цель: Научиться работать с файловым менеджером mc

Задание:

Умные люди утверждают, что 80% всей информации человек получает с помощью зрения. Недаром говорят, что «лучше один раз увидеть, чем сто раз услышать». Следствием этого положения является то, что при работе в терминале желательно пользоваться некоторым средством, которое визуально показывало бы текущее состояние. Иначе пользователю системы придётся помнить состояние и запоминать, что он делал, что делает, а это излишнее напряжение.

Одним из таких несложных средств, которое представляет информацию визуально и которое позволяет несколько автоматизировать работу пользователя, является mc (Midnight Commander). Но для удобства и эффективности работы это средство необходимо правильно настроить.

Для освоения файлового менеджера mc выполните следующие действия.

Итак, предполагается, что вы **работаете в текстовом (терминальном) режиме. Если вы ещё в графическом (в KDE, Gnome или где-то ещё), то перейдите в текстовый режим. Для этого нажмите одновременно клавиши Ctrl-Alt-F2**

1. В левом верхнем углу терминала видите приглашение вида

```
<имя_компа>  login:
```

2. Входим в систему: вводим логин и пароль.

Появляется приглашение вида:

```
[login@имя_компа ~]$
```

Это означает, что мы в системе. Мы вошли под именем <login> на компьютер <имя_компа>. Значок ~ после двоеточия означает, что мы находимся в своём домашнем каталоге, который называется /home/<login>. Для нас система запустила оболочку bash. Значок $ - приглашение оболочки вводить команды.

3. Вводим команду:

```
[login@имя_компа ~]$ mc
```

Нажимаем Enter. В результате выполнения команды должен запуститься файловый менеджер Midnight Commander. На рисунке 21 показано, как выглядит окно mc сразу после установки пакета и при первом запуске.

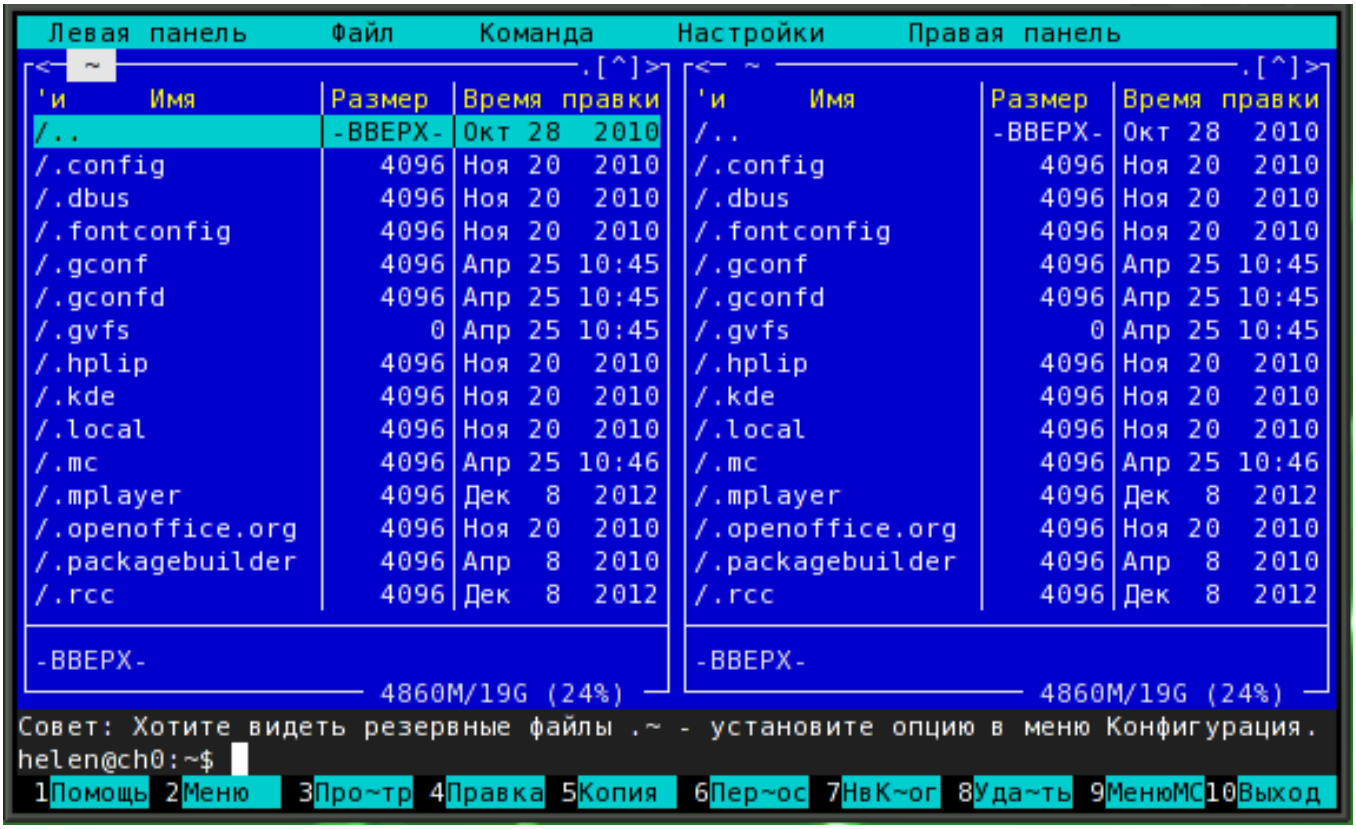


Рис. 21. Окно mc. Ненастроенное.

Самая верхняя строка окна — строка меню. Вход в меню осуществляется посредством нажатия функциональной клавиши **F9**, а выход («без сохранения») - двойное нажатие клавиши **Esc**. Поскольку в меню всегда можно попасть с помощью клавиши F9, то строку меню можно удалить.

Среднюю часть окна занимают две панели — левая и правая. В них отображается содержание текущих каталогов. Отображаться может один и тот же каталог или разные. Видно, что в левой панели на каталоге «**..**» установлена полоса-курсор (Bar). Она может передвигаться по каталогам и файлам с помощью клавиш «стрелка_вверх» и «стрелка_вниз», а между панелями Bar переключается клавишей Tab. Панель, в которой находится Bar, называется **активной.**

Нижнюю часть панелей занимает статусная строка («мини-статус»), в которой с левой стороны выводится имя каталога/файла, на котором установлен Bar (на рисунке 21 Bar установлен на каталоге «..», который обозначает вышестоящий (родительский) каталог, поэтому вместо имени каталога в статусе видим слово «ВВЕРХ»), а с правой стороны выводится итоговая информация по файловой системе в форме: <сколько_свободно>/<общий_размер_файловой_системы> и в круглых скобках — процент свободного места.

Следующая строчка начинается со слова «Совет:». Это «строка подсказки», которую обычно никто не читает. Поэтому её тоже можно удалить.

Следующая строчка — командная строка оболочки. В ней можно ввести команду оболочки (bash'a), нажать Enter и команда оболочки будет выполнена. Но результат выполнения команды

(вывод команды) мы не увидим, потому что сразу после завершения команды будет восстановлено окно mc. Чтобы увидеть результат выполнения команды, нужно нажать клавиши Ctrl-o, чтобы вернуться в mc нужно нажать Ctrl-o ещё раз.

И самая нижняя строка окна — ещё одна подсказка: в ней перечислены используемые в mc функциональные клавиши и их назначение:

F1 — вызов большой справки (online help);

F2 — меню пользователя; используется для запоминания некоторых часто используемых действий, оформленных в виде скриптов;

F3 — просмотр содержимого файла во встроенном просмотрщике; выход из просмотра — F3;

F4 — встроенный текстовый редактор, простой и удобный;

F5 — копирование выделенного Bar'ом файла из одной панели в другую;

F6 — перенос выделенного Bar'ом файла из одной панели в другую;

F7 — создание нового каталога;

F8 — удаление выделенного Bar'ом файла или каталога;

F9 — вход в меню mc;

F10 — выход из mc.

4. Рекомендуемые настройки mc.

4.1. Показывать экран терминала после выполнения команды. Для этого выполняем следующие действия: F9 → Настройки → Конфигурация. Открывается панель выбора «Параметров конфигурации» (см. рис. 22). На этой панели в разделе «Пауза после выполнения . . .» клавишами перемещения курсора (клавиши со стрелками на клавиатуре) выделить строку

( ) Всегда

и нажать клавишу пробел. Символ «*», который стандартно стоит в строке «На тупых терминалах», переместится в строку «Всегда».

Нажать клавишу «Enter».

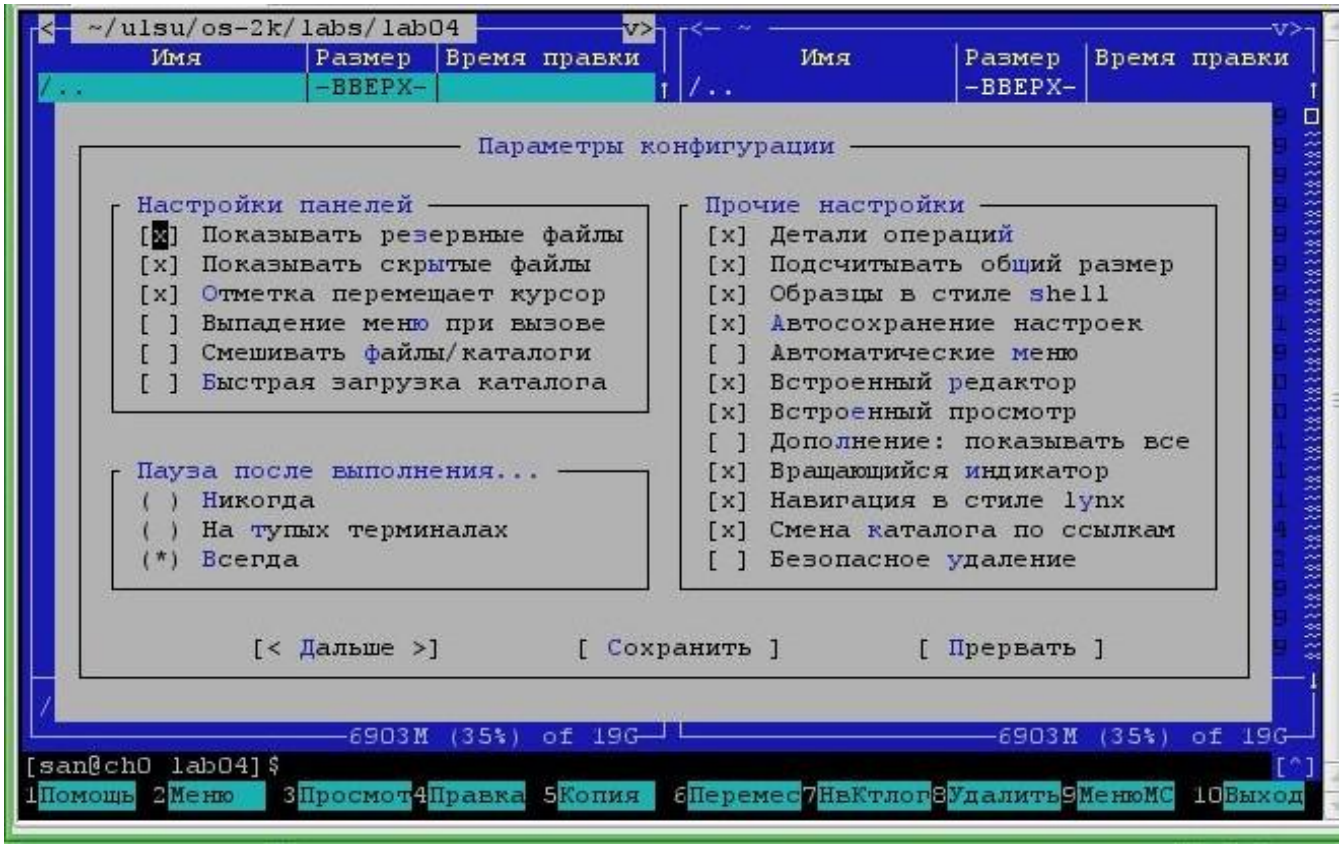


Рис. 22. Панель выбора параметров конфигурации — установка паузы после выполнения

4.2. Увеличиваем «жизненное пространство» - убираем линейку меню и строку подсказки. Меню всегда можно вызвать функциональной клавишей F9, а подсказки всё равно никто не читает. Для этого выполняем следующие действия: F9 → Настройки → Внешний вид. Открывается панель настройки внешнего вида (см. рис. 23). На этой панели в разделе «Прочие настройки» клавишей пробел убираем крестики в строках «Линейка меню» и «Строка подсказки». После чего нажимаем клавишу «Enter».

4.3. Сохраняем сделанные настройки: F9 → Настройки → Сохранить настройки (см. рис. 24). Настройки сохраняются в персональном конфигурационном файле пользователя, который находится в каталоге .mc (обратите внимание на то, что перед именем каталога стоит «.»).

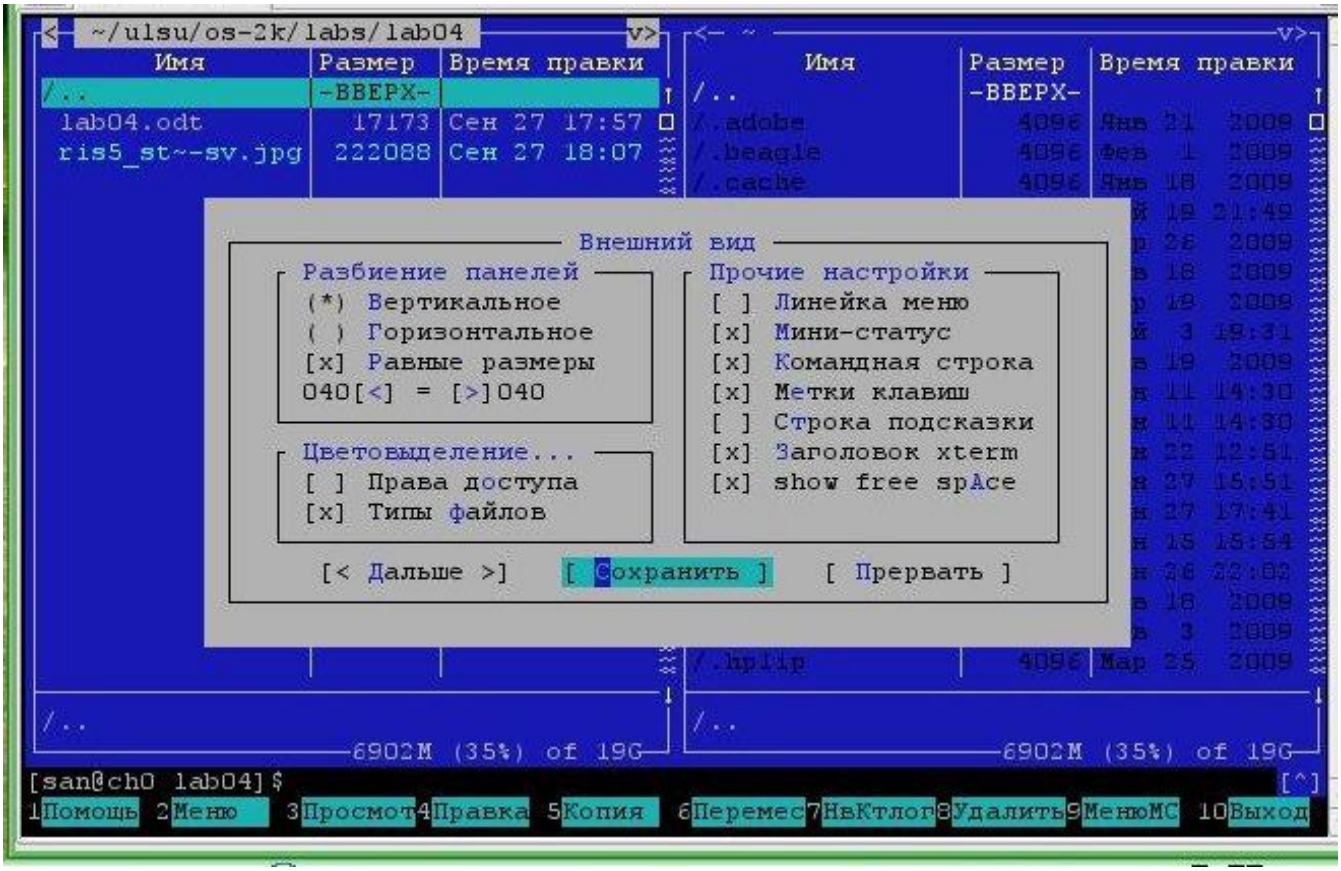


Рис. 23. Панель настройки внешнего вида.

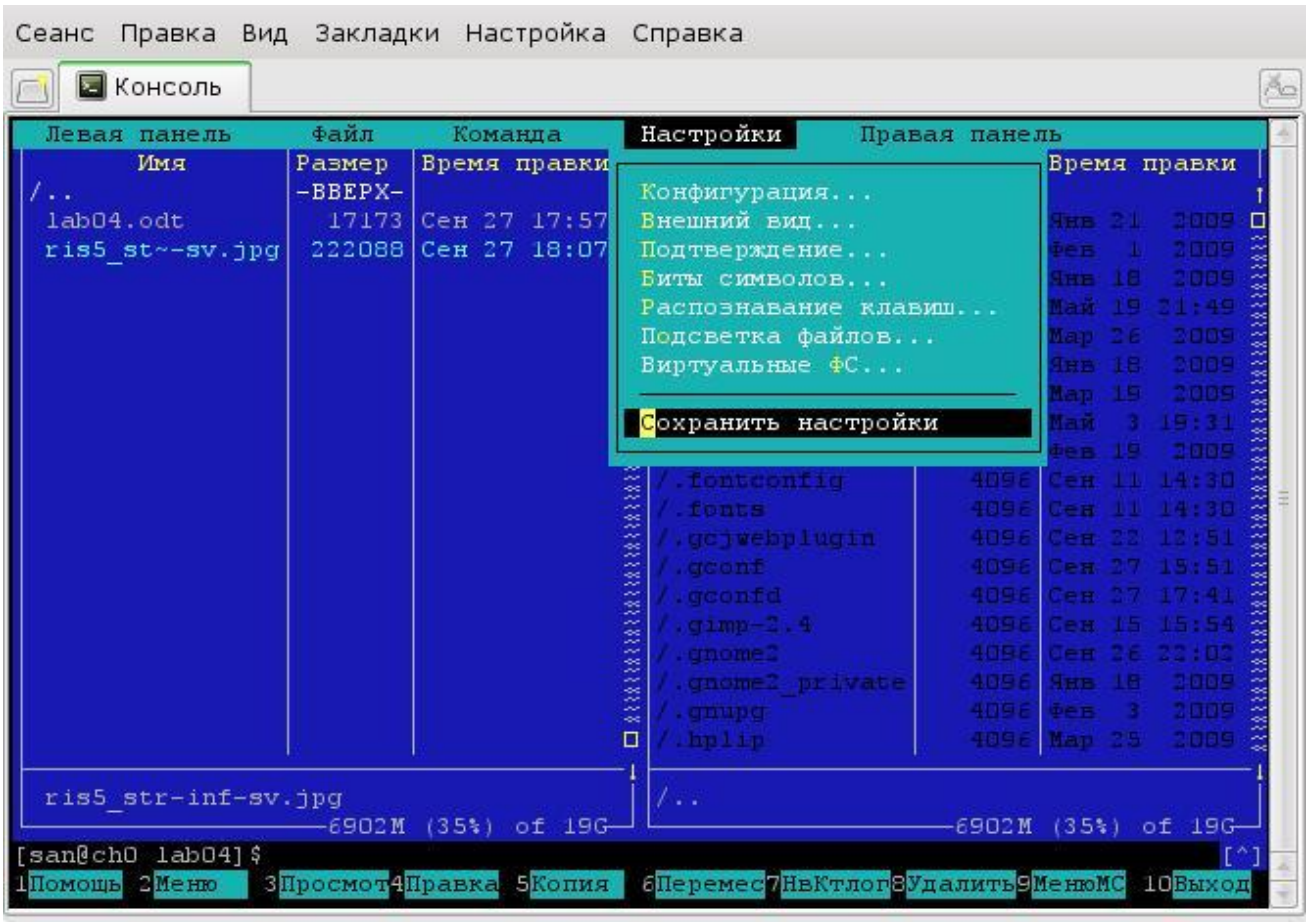


Рис. 24. Сохранение настроек.

4.4. Теперь окно mc выглядит так как на рисунке 25, то есть, за счёт исключения лишних элементов «жизненное пространство» увеличилось на две строки. Кроме того, после выполнения

команды оболочки, введённой в командной строке, окно mc восстанавливается не сразу, а только после нажатия any key, что позволяет увидеть вывод команды — результат её выполнения.

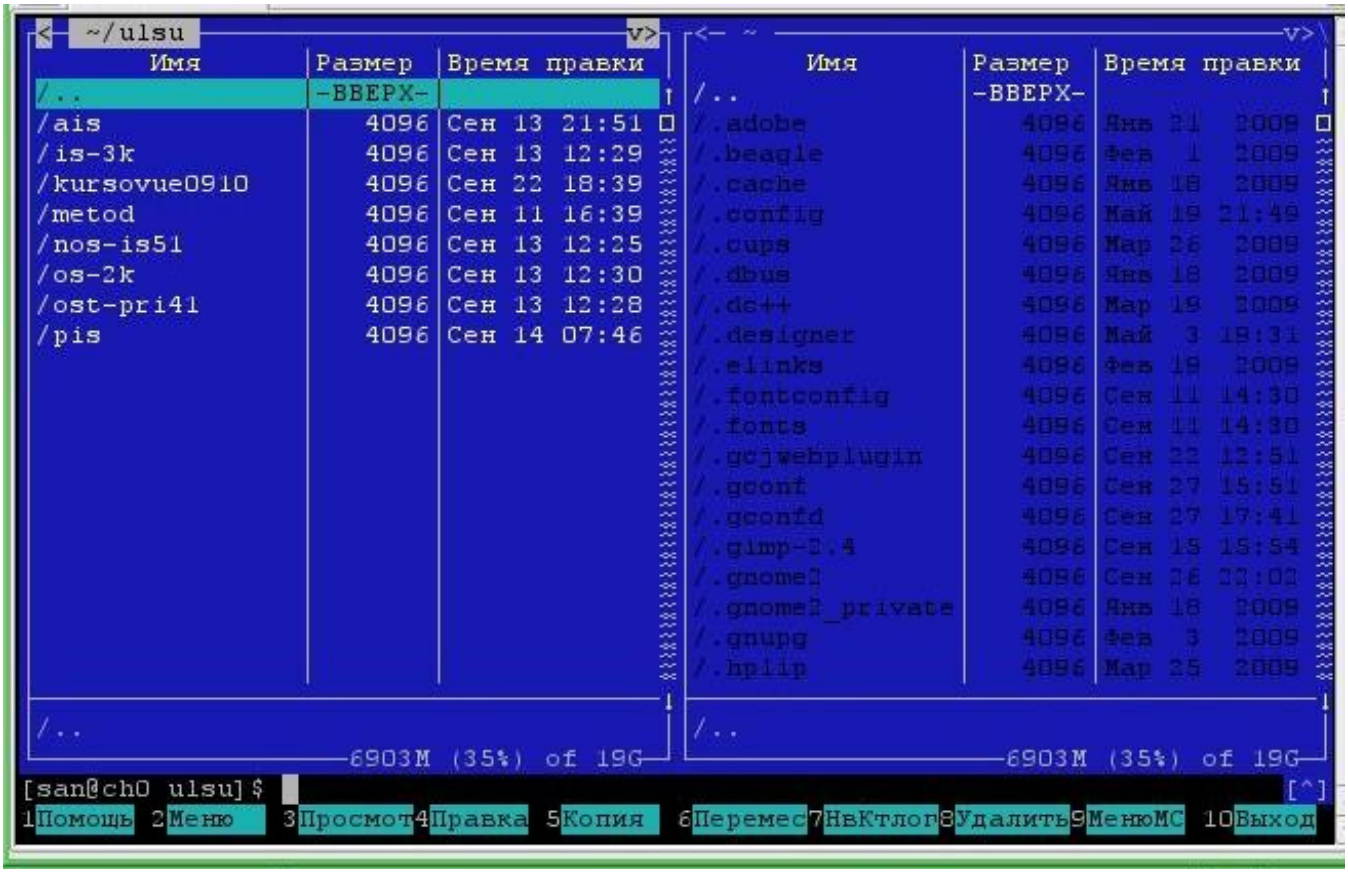


Рис. 25. Так окно mc должно выглядеть после произведённых настроек.

5. Справочная система mc. Справка вызывается функциональной клавишей F1. Читать. В системе русский файл справки расположен обычно здесь: /usr/share/mc/mc.hlp.ru.

Примечание-требование. Важное**.** Везде далее для имён каталогов и файлов использовать только латинские буквы, цифры и спецсимволы «-» - тире, «_» - подчёркивание. Все остальные символы — запрещены.

6. Выполнить в mc следующие действия:

6.1. Сделать активной левую панель; переключение между панелями — клавиша Таб;

6.2. Перейти в ней в **свой домашний каталог**; перемещение между файлами и каталогами осуществляется с помощью клавиш «стрелка вниз» и «стрелка вверх»;

6.3. Создать каталог <фамилия_латинскими_буквами_>; в конце имени каталога должен стоять знак подчёркивания; создание каталога — функциональная клавиша F7;

6.4. Сделать активной правую панель;

6.5. Перейти в ней в **свой домашний каталог**;

6.6. Создать каталог <фамилия_латинскими_буквами_1>;

6.7. Зайти в каталог <фамилия_латинскими_буквами_1>, то есть сделать его текущим;

6.8. Нажать Shift-F4, откроется окно встроенного редактора mc; ввести текст «Я, ФИО, выполняю лабораторную работу № 6»;

6.9. Сохранить файл с именем <фамилия_латинскими_буквами.txt>; сохранение файла во встроенном редакторе — функциональная клавиша F2;

6.10. Добавить в созданный файл текущую дату и время командой date;

6.11. Посмотреть содержимое созданного файла — функциональная клавиша F3; выход из режима просмотра — повторное нажатие клавиши F3;

6.12. Скопировать созданный файл в каталог <фамилия_латинскими_буквами_>; копирование файла — функциональная клавиша F5; **не забывайте: чтобы скопировать файл из одного каталога в другой, нужно открыть каталоги в обеих панелях, выделить нужный файл Bar'ом, нажать клавишу F5, нажать клавишу Enter;**

6.13. Найти в файловой системе файл с именем hosts (начинайте поиск от корневой файловой системы, а не из домашнего каталога); как искать — смотреть в справочной системе mc;

6.14. Скопировать файл hosts в каталог <фамилия_латинскими_буквами_1>;

6.15. Сделать копию файла hosts в файл hosts1, а файл hosts удалить; удаление файла — функциональная клавиша F8;

6.16. Командой cat добавить содержимое файла hosts1 в файл <фамилия_латинскими_буквами.txt>;

6.17. Командой cat добавить содержимое файла /etc/fstab в файл <фамилия_латинскими_буквами.txt>;

6.18. Добавить в файл <фамилия_латинскими_буквами.txt> текущую дату и время командой date;

6.19. Посмотреть что получилось в этом файле — клавиша F3.

Порядок сдачи лабораторной.

В отчёте должно быть:

а) задание на работу;

б) распечатка файла <фамилия_латинскими_буквами.txt>;

в) объяснение (комментарии) проделанной работы.

По требованию преподавателя повторить работу в лаборатории.

Дополнительная справочная информация

Краткие сведения о Midnight Commander'e

Здесь приведены краткие сведения о Midnight Commander'e, полный текст смотрите в файле /usr/share/mc/mc.hlp.ru.

Что такое Midnight Commander

Midnight Commander (mc) - это программа, предназначенная для просмотра содержимого каталогов и выполнения основных функций, управления файлами в Unix-подобных операционных системах. То есть, mc — это программа-оболочка, которая работает поверх shell и в некоторой степени автоматизирует работу пользователя.

Главное окно программы

Главное окно программы Midnight Commander состоит из трех полей. Два поля, называемые "панелями", идентичны по структуре и обычно отображают перечни файлов и подкаталогов каких-то двух каталогов файловой структуры. Эти каталоги в общем случае различны, хотя, в частности, могут и совпасть. Каждая панель состоит из заголовка, списка файлов и информационной строки.

Третье поле экрана, расположенное в нижней части экрана, содержит командную строку текущей оболочки. В этом же поле (самая нижняя строка экрана) содержится подсказка по использованию функциональных клавиш F1 - F10.

Самая верхняя строка экрана содержит строку «горизонтального меню» (Menu Bar). Эта строка может не отображаться на экране; в этом случае доступ к ней можно получить, щелкнув мышью по верхней рамке или нажав клавишу F9.

Панели Midnight Commander обеспечивают просмотр одновременно двух каталогов. Одна из панелей является активной в том смысле, что пользователь может выполнять некоторые операции с отображаемыми в этой панели файлами и каталогами.

В активной панели подсвечено имя одного из каталогов или файлов, а также выделен цветом заголовок панели в верхней строке. Этот заголовок совпадает с именем отображаемого в данной панели каталога, который является текущим каталогом той оболочки, из которой запущена программа. Вторая панель - пассивна. Почти все операции выполняются в активной панели, то есть в соответствующем (текущем) каталоге. Некоторые операции (типа копирования или переноса файлов) по умолчанию используют каталог, отображаемый в пассивной панели, **как место назначения операции.**

Вы можете выполнить любую команду операционной системы или запустить на исполнение любую программу непосредственно из программы Midnight Commander, просто набрав имя этой команды (программы) в командной строке и нажав клавишу Enter.

Поддержка мыши

Программа Midnight Commander обеспечивает поддержку мыши. Это свойство обеспечивается независимо от того, откуда запущен терминал (xterm или console) (даже если терминал запущен на удаленном компьютере, используя соединение через telnet, ssh или rlogin) или если вы работаете за консолью Linux и запущена программа управления мышью gpm.

Если вы щелкаете мышью на имени файла в одной из панелей, файл выбирается (подсветка перемещается на это имя); если вы щелкнете правой кнопкой мыши, файл отмечается (или отметка с файла снимается, в зависимости от предыдущего состояния).

Двойной щелчок мыши на имени файла означает попытку запустить файл на исполнение (если это исполняемая программа); либо, если «файл расширений» (Extension File Edit) содержит программу, ассоциированную с данным расширением, запускается эта программа и ей передается на обработку выбранный файл.

Точно также можно выполнить команду, ассоциированную с любой функциональной клавишей, щелкнув по соответствующей экранной кнопке в самой нижней строке экрана.

Если щелкнуть мышью по верхней рамке панели, отображающей очень длинный список файлов, происходит перемещение списка на одну колонку назад. Щелчок по нижней рамке панели приводит, соответственно, к перемещению по списку на целую колонку вперед. Этот метод перемещения работает также при просмотре «встроенной подсказки» (Help'a) и просмотре окна «Дерево каталогов» (Directory Tree).

По умолчанию скорость эмуляции повторных нажатий на клавишу в случае ее удержания (auto repeat rate) составляет 400 миллисекунд. Это значение можно изменить путем изменения параметра «mouse_repeat_rate» в файле ~/.mc/ini.

Если Commander запущен с поддержкой мыши, вы можете обойти Commander и добиться того, что мышь будет вести себя так же, как она ведет себя по умолчанию (обеспечивая вырезание и вставку текста), если будете удерживать клавишу Shift.

Некоторые команды

Если Midnight Commander скомпилирован с поддержкой подоболочки (subshell), вы можете в процессе выполнения приложения из-под Midnight Commander в любой момент набрать C-o и вернуться к главному экрану Midnight Commander-а.

Для возврата к вашему приложению достаточно снова набрать C-o. Если вы застопорите выполнение приложения, использовав этот прием, вы не сможете запустить других программ из Midnight Commander пока отложенное приложение не закончит работу, либо пока вы не прервете его выполнение.

Часто используемые команды:

- Tab (или C-i) Сменить текущую (активную) панель. Подсветка перемещается с панели, которая была активной ранее, в другую панель, которая становится активной.

- Insert (или C-t) Чтобы отметить файл, на который указывает в данный момент подсветка, используйте клавишу Insert (the kich1 terminfo sequence). Для снятия отметки с файла используются те же комбинации.

- Alt-t Циклически переключает режимы отображения списка файлов текущего каталога. С

помощью этой комбинации клавиш можно быстро переключаться из режима стандартного вывода (long listing) к сокращенному или к режиму, определяемому пользователем.

- + (plus) Эта клавиша используется для того, чтобы выбрать (отметить) группу файлов по регулярному выражению, задающему эту группу. Когда включена опция «Только файлы», то выделены будут только файлы. Если опция «Только файлы», отключена, то выделены будут как файлы, так и каталоги. Если включена опция «Образцы в стиле shell» (Shell Patterns), регулярные выражение строятся по тем же правилам, которые действуют в оболочке shell (* означает ноль или большее число любых символов, а ? заменяет один произвольный символ). Если опция «Образцы в стиле shell» (Shell Patterns) отключена, то пометка файлов производится по правилам обработки нормальных регулярных выражений (смотрите man ed). Если включена опция «С учётом регистра» то пометка файлов и каталогов будет производиться с учетом регистра символов имён. Если опция «С учётом регистра» отключена, то регистр символов учитываться не будет.

- \ (backslash) Клавиша "\" снимает отметку с группы файлов, то есть производит действие, обратное тому, которое вызывается по клавише "+".

- Page-Up (или C-p) Перемещает подсветку на предыдущую позицию в списке файлов панели.

- Page-Down (или C-n) Перемещает подсветку на следующую позицию в списке файлов панели.

- home (или Alt-<) Перемещает подсветку на первую позицию списка файлов.

- end (или Alt->) Перемещает подсветку на последнюю позицию списка файлов.

Командная строка оболочки

В этом разделе перечислены команды, которые позволяют сократить число нажатий на клавиши во время ввода и редактирования команд в командной строке:

- Alt-Enter Копирует подсвеченное имя файла или каталога в командную строку.

- C-Enter То же самое, что M-Enter, но работает только на консоли Linux.

- C-q Эта команда (the quote command) используется для того, чтобы вставить символы, которые каким-то образом интерпретируются самим Midnight Commander-ом (например, символы '+', '-', '*', '!' и другие; поэтому нельзя использовать эти и подобные символы в именах файлов).

Клавиши управления перемещением

Встроенная программа просмотра файлов, программа просмотра подсказки и программа просмотра каталогов используют один и тот же программный код для управления перемещением.

Следовательно, для перемещения используются одни и те же комбинации клавиш. Но в каждой подпрограмме имеются и комбинации, применяющиеся только в ней.

Другие части Midnight Commander'a тоже используют некоторые из комбинаций клавиш управления перемещением, так что настоящая секция руководства может быть также полезна при

изучении этих частей.

Клавиши управления перемещением:

- **«Стрелка вверх» (или C-p)** Перемещение на одну строку назад или вверх
- **«Стрелка вниз» (или C-n)** Перемещение на одну строку вперед
- **Page Up (или M-v)** Перемещение на одну страницу назад
- **Page Down (или C-v)** Перемещение на одну страницу вперед
- **Home** Перемещение к началу.
- **End** Перемещение к концу.

Редактирование строк ввода

При вводе команд в строку ввода используются некоторые управляющие комбинации клавиш. Они описаны в руководстве на mc.

Главное меню программы Midnight Commander

Строка главного меню появляется в верхней части экрана после нажатия клавиши F9 или щелчка мыши по верхней рамке экрана. Меню состоит из пяти пунктов: "Левая", "Файл", «Команды", Настройки" и "Правая" (в английской версии соответственно "Left", "File", "Command", "Options" и "Right"). При выборе одного из этих пунктов появляется соответствующее выпадающее меню. Все они подробно описаны в руководстве на mc.

Встроенная программа просмотра файлов

Встроенная программа просмотра файлов имеет два режима просмотра: режим ASCII и шестнадцатеричный (hex). Для переключения режимов используется клавиша F4. Если у вас установлена программа gzip проекта GNU, она будет использована для автоматического просмотра сжатых файлов.

Встроенная программа просмотра всегда пытается использовать для отображения информации лучший из методов, предоставляемых вашей системой для данного типа файла. Некоторые последовательности символов интерпретируются для задания таких атрибутов, как жирный шрифт и подчеркивание, обеспечивая более наглядное представление информации.

В шестнадцатеричном режиме функция поиска позволяет задать строку поиска как в обычном текстовом виде (заключенном в кавычки), так и в виде шестнадцатеричных констант. Можно даже одновременно использовать в шаблоне поиска как ту, так и другую форму представления, например:

"String" -1 0xBB 012 "more text"

Обратите внимание, что 012 является восьмеричным числом, -1 преобразовывается в 0xFF, а текст между кавычками и константами игнорируется.

Встроенный редактор

Встроенный редактор обеспечивает выполнение большинства функций редактирования, присущих полноэкранным редакторам текста. Он вызывается нажатием клавиши F4 при условии, что в инициализационном файле установлена в 1 опция use_internal_edit. Размер редактируемого файла не может превышать 16 Мегабайт. С помощью этого редактора можно редактировать двоичные файлы без потери данных.

Поддерживаются следующие возможности: копирование, перемещение, удаление, вырезание и вставка блоков текста; отмена предыдущих операций (key for key undo); выпадающие меню; вставка файлов; макроопределения; поиск и замена по регулярным выражениям; выделение текста по комбинации клавиш shift-стрелки в стиле MSW-MAC (только для Linux-консоли); переключение между режимами вставки-замены символа; а также операция обработки блоков текста командами оболочки (an option to pipe text blocks through shell commands like indent).

Редактор очень прост и практически не требует обучения. Для того, чтобы узнать, какие клавиши вызывают выполнение определенных действий, достаточно просмотреть выпадающие меню, которые вызываются нажатием клавиши F9 в окне редактора. Не перечисленные в меню комбинации клавиш:

- Shift-<клавиши стрелок> выделение блока текста.
- Ctrl-Ins копирует блок в файл cooledit.clip.
- Shift-Ins производит вставку последнего скопированного в cooledit.clip блока в позицию курсора.
- Shift-Del удаляет выделенный блок текста, запоминая его в файле cooledit.clip.
- По клавише Enter вставляются символы конца строки, причем на следующей строке автоматически устанавливается отступ.

Работает выделение текста с помощью мыши, причем если удерживать клавишу Shift, то управление мышью осуществляется терминальным драйвером мыши.

Для того, чтобы определить макрос, нажмите Ctrl-R, после чего введите строки команд, которые должны быть выполнены. После завершения ввода команд снова нажмите Ctrl-R и свяжите макрос с какой-нибудь клавишей или комбинацией клавиш, нажав эту клавишу (комбинацию). Макрос будет вызываться нажатием Ctrl-A и назначенной для него клавиши. Макрос можно также вызвать нажатием любой из клавиш Meta (Alt), Ctrl, или Esc и назначенной макросу клавиши, при условии, что данная комбинация не используется для вызова какой-либо другой функции. Макрокоманды после определения записываются в файл .mc/cedit/cooledit.macros в вашем домашнем каталоге. Вы можете удалить макрос удалением соответствующей строки в этом файле.

По клавише F19 (ее нет на обычной клавиатуре IBM PC, так что придется пользоваться

соответствующим пунктом меню, вызываемым по клавише F9, или переназначить клавишу) будет осуществляться форматирование выделенного блока кода на языке C, C++ или других. Форматирование управляется файлом /usr/share/mc/edit.indent.rc который при первом вызове копируется в .mc/cedit/edit.indent.rc в вашем домашнем каталоге.

Встроенный редактор обрабатывает символы из второй половины кодовой таблицы (160+). Но когда **редактируете бинарные файлы**, лучше установить опцию "Биты символов" (Display bits) из меню "Настройки" в положение "7 бит", чтобы сохранить формат файла (to keep the spacing clean).

Описать все функции встроенного редактора в данной подсказке невозможно. Запомните только, что все основные операции можно выполнить через пункты меню, которое вызывается нажатием клавиши F9 в окне редактирования. Кроме того, можно прочитать man-страницу по команде man mcedit (или info mcedit [Internal File Editor / options]).

Виртуальные файловые системы

Программа Midnight Commander содержит подпрограммы, обеспечивающие доступ к различным файловым системам. Эти подпрограммы (их совокупность называется переключателем виртуальных файловых систем - virtual file system switch) позволяют Midnight Commander-у манипулировать файлами, расположенными на не-Unix-овых файловых системах.

В настоящее время Midnight Commander обеспечивает поддержку нескольких Виртуальных Файловых Систем - ВФС (VFS):

- локальной файловой системы, используемой для обычных файловых систем Unix;

- файловой системы ftpfs, используемой для манипулирования файлами на удаленных компьютерах по протоколу FTP;

- файловой системы tarfs, используемой для обработки tar- и сжатых tar-файлов;

- файловой системы undelfs, используемой для восстановления удаленных файлов в файловой системе ext2 (файловая система, используемая в Linux по умолчанию);

- файловой системы fish (для манипулирования файлами при работе с оболочкой через такие программы как rsh и ssh);

- и, наконец, сетевой файловой системы nfs.

MC может быть собран с поддержкой файловой системы smbfs, используемой для манипулирования файлами на удаленных компьютерах по протоколу SMB (CIFS).

Подпрограммы работы с виртуальными файловыми системами интерпретируют все встречающиеся имена путей и формируют корректные обращения к различным файловым системам. Форматы обращения к каждой из виртуальных файловых систем описаны в отдельных разделах по каждой ВФС.

Файловая система ftpfs (FTP File System)

Файловая система ftpfs позволяет работать с файлами на удаленных компьютерах. Для этого можно использовать команду "FTP-соединение" (доступную из меню левой и правой панелей) или же непосредственно сменить текущий каталог командой cd, задав путь к каталогу следующим образом:

ftp:[!][user[:pass]@]machine[:port][remote-dir]

Элементы user, port и remote-dir не обязательны. Если элемент user указан, то Midnight Commander будет пытаться регистрироваться на удаленном компьютере с этим именем, в противном случае будет использовано имя anonymous или имя из файла ~/.netrc. Необязательный элемент pass#(если указан) используется как пароль для входа. Однако явно задавать его не рекомендуется (также не записывайте его в ваши hotlist, если только вы не обеспечили соответствующую защиту этих файлов; но и тогда нельзя быть полностью уверенным в безопасности).

Примеры:

ftp:ftp.nuclecu.unam.mx/Linux/local

ftp:tsx-11.mit.edu/pub/Linux/packages

ftp:!behind.firewall.edu/pub

ftp:guest@remote-host.com:40/pub

ftp:miguel:xxx@server/pub

Для того, чтобы соединиться с сервером, который расположен за firewall, нужно использовать префикс ftp:! (то есть добавить восклицательный знак перед именем сервера), чтобы указать Midnight Commander на необходимость использовать прокси для осуществления передач по ftp. Вы можете задать имя прокси в диалоговом окне «Виртуальные ФС...» (Virtual FS) меню "Настройки".

Чтобы не задавать имя прокси-сервера каждый раз, можно поставить отметку в квадратных скобках возле опции «Всегда использовать FTP прокси» в диалоговом окне «Виртуальные ФС...» (Virtual FS) меню "Настройки". В таком случае программа всегда будет использовать указанный прокси-сервер. При этом (если опция установлена) программа делает следующее: считывает из файла /usr/share/mc/mc.no_proxy имена локальных машин (если имя начинается с точки, оно считается именем домена), и, если заданное при установлении FTP-соединения имя машины совпадает с одним из имен, указанных в файле mc.no_proxy без точки, то производит прямое обращение к данной машине.

При подключении к ftp-серверу через фильтрующий пакеты маршрутизатор (If you are using the ftpfs code with a filtering packet router), который не позволяет использовать обычный режим открытия файлов, можно заставить программу работать в режиме пассивного открытия файла (the

passive-open mode). Для этого установите в инициализационном файле опцию ftpfs_use_passive_connections в 1.

Midnight Commander сохраняет в течение заданного интервала времени список файлов удаленного каталога, прочитанный по FTP, в оперативной памяти. Величина этого интервала времени задается в диалоговом окне «Виртуальные ФС...» (Virtual FS) меню "Настройки". В силу этого возможен побочный эффект, заключающийся в том, что даже если вы сделали какие-то изменения в каталоге, они не будут отображаться в панели до тех пор, пока вы не обновите содержимое панели командой C-r. Это не является недоработкой (если вы думаете, что это ошибка, поразмыслите над тем, как происходит работа по FTP с файлами, находящимися на другой стороне Атлантического океана).

Файловая система FISH (FIle transfer over SHell)

Файловая система fish — это сетевая файловая система, которая позволяет работать с файлами на удаленном компьютере так, как если бы они были расположены на вашем диске. Для того, чтобы это было возможно, на удаленном компьютере должен быть запущен fish-сервер, или bash-совместимая оболочка shell.

Для соединения с удаленным компьютером нужно выполнить команду перехода в каталог (chdir), имя которого задается в следующем формате:

sh:[user@]machine[:options]/[remote-dir]

Элементы user, options и remote-dir не обязательны. Если задан элемент user, то Midnight Commander будет регистрироваться на удаленный компьютер под этим именем, в противном случае - под тем именем, с которым вы зарегистрированы в локальной системе.

В качестве options могут использоваться:

'C' - использовать сжатие;

'r' - использовать rsh вместо ssh;

port - использовать данный порт для поключения к удалённому компьютеру.

Если задан элемент remote-dir, то указанный каталог станет текущим после соединения с удаленным компьютером.

Примеры:

sh:onlyrsh.mx:r/Linux/local

sh:joe@want.compression.edu:C/private

sh:joe@noncompressed.ssh.edu/private

sh:joe@somehost.ssh.edu:2222/private

Файловая система NFS (Network File System)

Файловая система mc - это еще одна сетевая файловая система, которая позволяет работать

с файлами на удаленном компьютере. Для того, чтобы можно было воспользоваться этой ФС, на удаленном компьютере должна быть запущена серверная программа mcserv.

Для соединения с удаленным компьютером нужно выполнить команду перехода в каталог, имя которого строится в соответствии со следующим форматом:

mc:[user@]machine[:port][remote-dir]

Элементы user, port и remote-dir не обязательны.

Если задан элемент user, то Midnight Commander будет регистрироваться на удаленный компьютер под этим именем, в противном случае — под тем именем, с которым вы зарегистрированы в локальной системе.

Элемент port используется в том случае, если удаленный компьютер использует специальный порт (чтобы узнать, что такое порт и как его использовать, смотрите страницу руководства man mcserv. Если задан элемент remote-dir, то указанный каталог станет текущим после соединения с удаленным компьютером.

Примеры:

mc:ftp.nuclecu.unam.mx/Linux/local

mc:joe@foo.edu:11321/private

Файловая система UFS (Undelete File System)

В ОС Linux можно сконфигурировать файловую систему ext2fs, используемую по умолчанию, таким образом, что появится возможность восстанавливать удаленные файлы (но только в файловой системе ext2). Файловая система UFS (Undelete File System) представляет собой интерфейс к библиотекам ext2fs, позволяющий восстановить имена всех удаленных файлов, выбрать некоторое количество таких файлов и восстановить их.

Для того, чтобы воспользоваться этой возможностью (этой файловой системой), нужно выполнить команду перехода (chdir) в специальный каталог, имя которого образуется из префикса "undel" и имени специального файла устройства, на котором находится реальная файловая система.

Например, чтобы восстановить удаленные файлы на втором разделе первого SCSI-диска, нужно использовать следующее имя:

undel:sda2

Загрузка списка удаленных файлов требует некоторого времени, так что наберитесь терпения. Имейте в виду, что имена файлов в полученном списке будут цифровыми, так что поиск нужного придется проводить либо по дате, либо последовательным просмотром содержимого.

Файловая система smbfs

Файловая система smbfs позволяет работать с файлами на удаленных компьютерах по

протоколу SMB (CIFS) (Windows for Workgroups, Windows 9x/ME/XP, Windows NT, Windows 2000 и Samba). Для этого можно использовать пункт "SMB связь..." (доступный из меню левой и правой панелей) или же непосредственно сменить текущий каталог командой cd, задав путь к каталогу следующим образом:

smb:[username@]machine[/service][/remote-dir]

Элементы username, service и remote-dir необязательны. username, domain и password могут быть указаны в окне диалога.

Примеры:

smb:machine/Share

smb:other_machine

smb:guest@machine/Public/Irlex

Лабораторная работа № 8

Тема: УПРАВЛЕНИЕ ПРОЦЕССАМИ

Цель: Научиться работать с процессами из терминала

Задание:

Для освоения команд работы с процессами выполните следующие действия.

Итак, предполагается, что вы **работаете в текстовом (терминальном) режиме. Если вы ещё в графическом (в KDE, Gnome или где-то ещё), то перейдите в текстовый режим. Для этого нажмите одновременно клавиши Ctrl-Alt-F2**

1. В левом верхнем углу терминала видите приглашение вида

```
<имя_компа>  login:
```

2. Входим в систему: вводим логин и пароль.

Появляется приглашение вида:

```
[login@имя_компа ~]$
```

Прочтите пункт 2 задания на лабораторную работу 2.

Далее в задании будет опускаться содержимое квадратных скобок ([login@имя компа ~]), а будет указываться только символ «$».

Внимание: При сдаче лабы возможно придётся отвечать на вопросы о назначении и смысле команд. Мануал по команде:

$ man <команда> <Enter>

В том числе можно получить справку и по самой системе man:

$ man man <Enter>

3. Прежде всего, убедитесь, что вы находитесь в своём домашнем каталоге. Это можно сделать командой:

$ pwd

Если вы не в домашнем каталоге, то перейти в домащний каталог.

4. Создать файл с именем = фио (например, obama_b.txt) командой

$ **touch** <имя_файла>

5. Ввести в этот файл следующую информацию «Я, <фамилия имя отчество>, группа <группа>, лабораторная №8».

6. Добавить в этот файл две пустых строки.

7. Добавить в этот файл вывод команды **ps** так, чтобы были видны **id** пользователя, запустившего

процесс, **id** процесса, **id** родительского процесса, приоритет процесса, использование памяти процессом, использование CPU процессом, терминал процесса, команда запуска процесса.

8. Добавить в файл отчёта две пустых строки.

9. Добавить в файл отчета информацию о процессах запущенных пользователем root. Вывод должен быть отсортирован по номеру процесса. Как это сделать, смотри man ps.

10. Добавить в файл отчета информацию о процессах запущенных пользователем student так, чтобы были видны **id** пользователя, запустившего процесс, **id** процесса, **id** родительского процесса, приоритет процесса, использование памяти процессом, использование CPU процессом, терминал процесса, команда запуска процесса.

Вывод должен быть отсортирован по номеру процесса. Выборку процессов, принадлежащих пользователю student, делать с помощью команды grep. Сортировку вывода можно сделать с помощью программы sort.

11. Перейти на другой terminal (командой Alt-F3). Ввести команду top так, чтобы контролировать только процессы пользователя student (см. man top).

12. Вернуться во второй терминал. В этом терминале продемонстрировать работу команды kill — найти процесс top, запущенный в третьем терминале и убить его. Вывести результат в отчёт.

13. Вывести в отчёт результат выполнения команд

tty

w

uname -a

uptime

14. Добавить в этот файл дату командой «date».

Порядок сдачи лабораторной работы.

В отчёте должно быть:

а) задание на работу;

б) распечатка созданного файла с именем = fio;

в) объяснение (комментарии) проделанной работы.

По требованию преподавателя повторить работу в лаборатории и объяснить, что, собственно, делал.

Дополнительная справочная информация

Управление процессами

1. Процессы

Процесс (process) — совокупность области адресного пространства, в котором выполняется запущенная программа, и PCB — блока управления процессом. Любой процесс может запускать

другие процессы. То есть, есть процессы родительские и есть процессы — потомки. Таким образом, процессы в среде Unix образуют иерархическую структуру. На вершине этой структуры находится процесс init, являющийся предком всех остальных процессов.

1.1. Атрибуты процессов

С каждым процессом связан набор атрибутов, которые помогают системе контролировать выполнение процессов и распределять между ними ресурсы системы.

Идентификатор процесса (process ID) - это целое число, однозначно идентифицирующее процесс. Процесс с идентификатором 1 это процесс init.

Идентификатор родительского процесса (parent process ID) - указывает на родительский процесс.

Идентификатор группы процессов (process group ID) - процессы могут объединяться в группы; каждая группа обозначается идентификатором группы; процесс, идентификатор которого совпадает с идентификатором группы, называется лидером группы.

Идентификатор сеанса (session ID) - каждая группа процессов принадлежит к сеансу; сеанс связывает процессы с управляющим терминалом; когда пользователь входит в систему, все создаваемые им процессы будут принадлежать сеансу, связанному с его текущим терминалом.

Программное окружение (programm environment) - это просто набор строк, заканчивающихся нулевым символом; строки называются переменными окружения и имеют следующий формат:

имя переменной = значение переменной

Дескрипторы открытых файлов - дескриптор файла — некоторое число, которое используется для обращения к файлу; при запуске процесс наследует дескрипторы от родительского процесса.

Текущий рабочий каталог - это каталог от которого система производит разрешение относительных имен.

Текущий корневой каталог - это каталог от которого производится разрешение абсолютных имен; процесс не имеет доступа к файлам находящимся выше корневого каталога; Это свойство используется при создании «песочниц» для процессов.

Идентификаторы пользователя и группы - с каждым процессом связаны действительные идентификаторы пользователя (real user ID) и группы (real group ID), совпадающие с соответствующими идентификаторами пользователя, запустившего процесс; кроме того, с процессом связаны эффективные идентификаторы пользователя (effective user ID) и группы, определяющие права процесса в системе; обычно, действительные и эффективные идентификаторы совпадают.

Приоритет (nice) - значение nice ("дружелюбность") показывает готовность процесса уступить свое процессорное время другим процессам; чем больше значение nice, тем ниже приоритет процесса.

2. Основные сведения о работе с процессами

2.1. Создание процессов — вызовы fork и exec

Основным средством для создания процессов является системный вызов **fork.** При выполнении данного вызова ядро создает новый процесс, который является копией процесса, вызвавшего fork. Созданный процесс называется дочерним, а процесс осуществивший вызов fork — родительским. В дочернем процессе вызов возвращает значение ноль, а в родительском он возвращает идентификатор дочернего процесса. Дочерний процесс наследует дескрипторы открытых файлов и значения переменных окружения родительского процесса.

Другой системный вызов для работы с процессами — **exec.** Он позволяет сменить выполняемую программу. Вызову exec передаются в качестве аргументов имя программы, которую надо выполнить и список ее аргументов. При выполнении вызова в пространство памяти вызывающего процесса загружается новая программа, которая запускается с начала. При выполнении вызова exec дескрипторы открытых файлов сохраняют свое значение.

2.2. Завершение процессов

Для завершения процесса используется системный вызов **exit.** Вызов имеет целочисленный аргумент, называемый кодом завершения процесса. Как правило при успешном завершении процесса код завершения равен нулю, а в случае возникновения ошибки отличен от нуля. Родительский процесс может получить статус завершения дочернего процесса выполнив системный вызов **wait** или **waitpid**. Если родительский процесс завершается раньше дочернего (не сделав вызов wait или waitpid), то дочерний процесс по завершению переходит в состояние **зомби,** когда он выполняться уже не может, а ресурсы занимает.

3. Механизмы межпроцессного взаимодействия

Unix имеет большое число механизмов межпроцессного взаимодействия. Наиболее популярными средствами являются сигналы, программные каналы (pipes), именованные каналы (FIFO), shared memory, сообщения и сокеты. Особенно большое распространение получили последние. Большинство современного распределённого программного обеспечения используют для взаимодействия сокеты, информация о которых приводится в третьей части пособия.

Средства взаимодействия могут иметь локальный характер (то есть, могут передавать информацию только между процессами, запущенными в одной операционной системе) и глобальный (то есть, могут передавать информацию между процессами, запущенными в разных операционных системах).

3.1. Сигналы

Сигналы обеспечивают простой метод прерывания работы процессов. Сигналы

используются в основном для обработки исключительных ситуаций. Процесс может определять действия выполняемые при поступлении сигнала, блокировать сигналы, посылать сигналы другим процессам. Существует более двадцати различных сигналов. Основные:

SIGCHLD - сигнал о завершении дочернего процесса.

SIGHUP - сигнал освобождения линии. Посылается всем процессам, подключенным к управляющему терминалу при отключении терминала. Многие демоны при получении данного сигнала заново просматривают файлы конфигурации и перезапускаются.

SIGINT - сигнал посылается всем процессам сеанса, связанного с терминалом, при нажатии пользователем клавиши прерывания (CTRL-C).

SIGTERM - сигнал приводит к немедленному прекращению работы получившего сигнал процесса.

SIGKILL - сигнал приводит к немедленному прекращению работы получившего сигнал процесса. В отличие от SIGTERM процесс не может блокировать и перехватывать данный сигнал.

SIGSEGV - сигнал посылается процессу, если тот пытается обратиться к неверному адресу памяти.

SIGSTOP - сигнал приводящий к остановке процесса. Для отправки сигнала SIGSTOP активному процессу текущего терминала можно воспользоваться комбинацией клавиш (CTRL-Z).

SIGCONT - сигнал возобновляющий работу остановленного процесса.

SIGUSR1,SIGUSR2 - сигналы определяемые пользователем.

Для того, чтобы отправить процессу сигнал можно использовать команду **kill**. Для того, чтобы процесс мог отправить сигнал другому процессу необходимо чтобы эффективные идентификаторы пользователя у посылающего процесса и у процесса получателя совпадали. Процессы с эффективным идентификатором пользователя равным нулю могут посылать сигналы любым процессам.

3.2. Каналы

Часто возникает ситуация когда два процесса последовательно обрабатывают одни и те же данные. Для обеспечения передачи данных от одного процесса к другому в подобных ситуациях используются программные каналы. Программный канал (**pipe**) служит для установления связи, соединяющей один процесс с другим. Запись данных в канал и чтение из него осуществляются при помощи системных вызовов write и read, т.е. работа с каналами аналогична работе с файлами. Для создания программного канала используется системный вызов **pipe**. Вызов возвращает два дескриптора файлов, первый из которых открыт для чтения из канала, а второй для записи в канал.

Каналы используются, например, при организации конвейера. При выполнении команды:

find /usr/bin -name a* | sort

создается канал, команда find выводит в него результаты своей работы, а команда sort считывает из этого канала данные для сортировки.

Главным недостатком программных каналов является то, что они могут использоваться только для связи процессов имеющих общее происхождение (например, родительский процесс и его потомок). Другой недостаток - ограниченное время существования канала (программные каналы уничтожаются после завершения обращающегося к ним процесса).

Именованные каналы идентичны программным в отношении записи и чтения данных, но они являются объектами файловой системы. Именованный канал имеет имя, владельца и права доступа. Открытие и закрытие именованного канала осуществляется как открытие и закрытие любого файла, но при чтении и записи он ведет себя аналогично каналу.

Для создания именованного канала используется команда **mkfifo**. Если некоторый процесс открывает именованный канал для записи, то этот процесс блокируется до тех пор, пока другой процесс не откроет этот канал для чтения, и наоборот.

4. Команды для работы с процессами

ps [-axewjlu] [-o формат] [-U пользователь] [-p pid]

Выводит список и статус процессов работающих в системе. Без аргументов выводит список процессов текущего пользователя, подключенных к терминалу. Значения параметров следующие:

-a вывести информацию о процессах всех пользователей.

-x вывести информацию о процессах не подключенных к терминалу.

-e вывести значения переменных окружения процесса.

-w использовать строки длиной 132 символа. Если указан несколько раз, то строки не обрезаются совсем.

-j, -l, -u меняют формат вывода информации.

-o формат вывести информацию в указанном формате.

-U пользователь вывести информацию о процессах указанного пользователя.

-p pid вывести информацию о процессе с указанным идентификатором.

Значение формата для параметра -o является списком из следующих ключевых слов разделенных запятыми (без пробелов):

command - командная строка и аргументы.

nice - уровень nice (приоритет).

pgid - идентификатор группы процессов.

pid - идентификатор процесса.

ppid - идентификатор родительского процесса

rgid, ruid - реальные идентификаторы группы и пользователя.

uid - реальный идентификатор пользователя.

tty - управляющий терминал

Для различных систем параметры и ключевые слова могут сильно различаться. Подробности об использовании ps на конкретной системе можно получить при помощи команды man ps.

kill [-s signal| -signal] pid

Посылает сигнал указанному процессу. Если значение сигнала опущено, предполагается SIGTERM. signal — символическое имя сигнала без префикса SIG, либо номер сигнала. Пример: kill -HUP 172 — послать сигнал SIGHUP процессу с идентификатором 172.

nice [-nice] команда [аргументы]

Выполняет команду с меньшим (уменьшенным, пониженным) приоритетом. Если nice не задан, то предполагается 10. Значение nice может быть от -20 (наивысший приоритет) до 20 (наименьший приоритет). Отрицательные числа задаются как –nice. Увеличение приоритета может выполнить только суперпользователь.

Пример:

nice -10 john users — запустить программу john с пониженным приоритетом.

mkfifo [-m режим_доступа] имя

Создает именованный канал с указанным именем и режимом доступа.

tty

Выводит имя текущего терминала.

who [am i]

Выводит список пользователей работающих в системе.

uname [-amnrsv]

Выводит информацию о системе.

uptime

Выводит время работы системы и ее среднюю загрузку за последние 5, 10 и 15 минут.

5. Средства оболочки предназначенные для работы с процессами

Список — последовательность из одного или более конвейеров, разделенных операторами ;, &, && или ||. Более высокий приоритет у операторов && и ||. Если команда завершается

оператором &, то оболочка выполняет ее в фоновом режиме. Если между двумя командами стоит оператор &&, то вторая команда будет выполнена только в том случае, если первая завершится успешно. Если между двумя командами стоит ||, то вторая команда будет выполнена только в том случае, если код завершения первой команды отличен от нуля. Если команды разделены точкой с запятой, то вторая команда будет выполнена после завершения первой, независимо от результата выполнения первой команды.

Оболочка содержит несколько встроенных команд для работы с процессами:

wait [pid]

Ожидает завершения процесса с указаным идентификатором. Если идентификатор опущен, то ожидает завершения всех процессов запущеных оболочкой.

exec команда [аргумент]...

Указанная команда заменяет оболочку и получает в качестве параметров заданные аргументы.

exit [n]

Приводит к завершению оболочки с кодом завершения n. Если арумент опущен, то код завершения ноль.

trap [действие условие...]

Устанавливает обработчик события. Условие либо EXIT, либо имя сигнала без префикса SIG. EXIT соответствует завершению работы оболочки. Если действие равно “-”, то обработчик сбрасывается в значение по умолчанию. Например, после выполнения команды:

trap "echo PRESSED" INT

оболочка будет выводить слово PRESSED после каждого нажатия клавиш CTRL-C. (Нажатие клавиш CTRL-C приводит к посылке сигнала SIGINT процессам подключенным к терминалу).

Лабораторная работа № 9

Тема: УСТАНОВКА LINUX НА FLASH-НОСИТЕЛЬ

Цель: Научиться устанавливать ОС Linux на flash-носитель

Задание:

Установить ОС Linux на flash-носитель.

Порядок сдачи лабораторной работы

1. Продемонстрировать работу с установленной на флэшке системой:
   - загрузиться с флэшки,
   - выполнить на системе действия, по указанию преподавателя (как минимум, настройка Интернета и работа с hdd программой fdisk),
2. Представить отчет, в котором должно быть:
   а) задание на работу;
   б) описание процесса установки системы на flash-диск;
   в) краткое описание возможностей установленного на флэшку дистрибутива.

Указания к выполнению работы

1. Наличие загрузочной флэшки — необходимый атрибут системного/сетевого администратора. Но и для обычного продвинутого пользователя она также необходима, поскольку достаточно часто возникают задачи, требующие загрузки системы с другого носителя. Flash-диск в силу миниатюрности и достаточно большого объёма памяти вполне подходят для создания внешнего загружаемого устройства.

Для подобной установки есть специально созданные дистрибутивы, как правило, небольшого объёма: puppy — 130 Mb, frenzy — 200-250 Mb, DSL ~ 50-70Mb, feather ~ 120Mb и другие. Некоторые из них специально созданы как средства системного администратора.

2. Рекомендуемый дистрибутив — puppy (сайт www.puppyrus.org). Почему: при загрузке автоматически распознаются все виндовые и Linux'овые разделы (нескрытые, которые реально можно смонтировать), наличествующие на винчестере ПЭВМ и их иконки выводятся на Desktop.

Рекомендуемая версия — puppy-ГГ.ММ, где ГГ — год текущий, ММ — месяц. Это последняя (самая свежая) русская версия; на ftp-сервере можно видеть много разных версий puppy, которые суть либо адаптации английских версий, в работе менее удобных, либо несколько устарели.

Puppy устанавливается на файловую систему FAT32, то есть, после установки флешка остаётся в прежнем состоянии, только на неё добавляется ещё четыре файла, общим объёмом ~ 140 Мбайт.

3. Руководство по установке имеется на сайте http://docs.puppyrus.org/setups/start в разделе USB Flash.

Основная идея установки. Puppy рекомендуется ставить методом, так называемой, frugal-установки. Смысл её в том, что на флешку копируются три файла:

vmlinuz — ядро операционной системы, 2-3 Мб;

initrd.gz — образ RAM-диска, сжатый архив, который при запуске распаковывается в область оперативной памяти, организуемой как раздел диска; в архиве — нужное для для работы ядра системное ПО; объём — несколько десятков Мб;

pup_xxx.sfs — это тоже архив, содержащий прикладное ПО — те самые программы, которые будут видны из главного меню.

Процесс копирования этих трёх файлов как раз и организует «Универсальный инсталлятор Puppy» при установке puppy на флешку. Это вполне можно сделать вручную, выделив эти файлы из iso-образа puppy. На флешке должна быть файловая система FAT32 (как правило, так и есть) или файловая система EXT-2 («ну, это вряд ли»). После копирования этих файлов, останется...

4. Установить загрузчик. Рекомендуется grub-2 и рекомендуется это сделать из установленного puppy. В главном меню выбираем пункт «Конфигурирование загрузчика Grub4dos», в открывшемся окне указываем флешку (не ошибитесь!) и ждём примерно минуту. Иногда система спрашивает, не хотите ли что-нибудь подправить в конфигурационном файле. Если плохо соображаете, то не стоит ничего подправлять («а пусть она . . .»).

Но можно поставить и другой загрузчик, например, стандартный CD-шный sysLinux. Разные варианты описаны на сайте http://docs.puppyrus.org/setups/start

5. Но можно поставить на Flash и обычный дистрибутив. Например, AltLinux (Центр управления системой → Система → Создание загружаемого usb устройства). Однако это потребует флешки большого объёма и придётся создавать под Линукс отдельный раздел на флешке. Не рекомендуется. И вообще, большие дистрибутивы с флэшки работают медленно!

Дополнительная справочнаяинформация

Виды установки: FULL vs FRUGAL

Установка **FULL** — полная установка. Всех сбивает с толку название «полная», но в случае с Puppy это уступка для особо слабых машин, особенно с малым объемом оперативной памяти, когда из-за свопирования машина начинает заметно тормозить. Реально скорость **FULL** выше всего лишь приблизительно на 20%, зато к ней, и именно к ней, справедливы упреки в

небезопасности постоянной работы под root-ом в графическом режиме. Кроме того, при **FULL**-установке отсутствует /initrd, который является точкой монтирования для sfs-модулей, что приводит к необходимости их ручной распаковки и установки, а удаление установленных таким образом программ — очень сложный процесс.

Установка **FRUGAL** — ошибочно переводится как «формальная», более точно будет «упрощённая», «лёгкая». Это относится к легкости процедуры установки Puppy на жесткий диск таким методом, которая сводится к копированию трех-пяти файлов (зависит от версии), а не к работе установленной таким методом системы.

Фактически при такой установке происходит эмулирование загрузки с LiveCD, что для Puppy является основным режимом работы. **FRUGAL**-установка обеспечивает:

1. Работу с sfs-модулями, как постоянно подключенными, так и «на одну сессию», так называемое «горячее подключение».
2. Обеспечивает повышенную безопасность, так как sfs-файлы, будучи архивами, подключаются к системе «только для чтения» (ro), что исключает повреждение их содержимого случайными действиями пользователя. Такой файл можно только намеренно переименовать или удалить, но и восстановить не составит труда. Просто копируем на место удаленного sfs его «эталон» с CD.
3. Обеспечивает легкий бэкап, так как все изменения в системе хранятся в pup_save.2fs, то его можно просто скопировать в другое место или под другим именем, и в случае серьезного сбоя заменить «испорченный» save на «дубликат».

Для этого существует опция загрузки pfix=ram, которая добавляется в строку kernel файла конфигурации menu.lst загрузчика grub, при загрузке с LiveCD — в нижнюю строку загрузочного меню boot: (здесь пишется puppy pfix=ram). После этого происходит загрузка «с чистого листа» и можно проводить «восстановительные работы».

Для экономии места можно копировать только содержимое save-файла

```
cp -r /initrd/pup_rw /mnt/hdaN/savedir
```

Правда такой метод усложняет восстановление, так как вместо простой замены файла нужно заменить его содержимое, а для этого «неисправный» save надо примонтировать, очистить и скопировать сохраненное из savedir. Этот метод оправдан только при малом объеме жесткого диска.

Существует еще метод обеспечения безопасности — создание собственного sfs. Для этого достаточно скопировать содержимое /initrd/pup_ro2 в отдельно созданную директорию (например root-dir), «наложить» сверху содержимое /initrd/pup_rw и создать свой sfs командой

```
mksquashfs root-dir pup_301-mydisk.sfs
```

После создания sfs (процесс не быстрый) заменить им «штатный» sfs. Тогда необходимость в pup_save.2fs и zdrv-301.sfs отпадает.

Два замечания. Копирование лучше производить в графическом режиме (мышкой), почему-то меньше ошибок. И новый sfs будет пытаться стартовать в консоли, при первом запуске точно потребуется команда xwin, но это решаемо.

Также при **FRUGAL**-установке можно сделать минимального размера save-файл с самыми необходимыми настройками. Его легко вернуть «на родину» после краха и увеличить размер при необходимости. А потом просто кликнуть на старом save-файле (с другим именем), примонтировав таким образом, и скопировать оттуда необходимые настройки, которые обычно находятся в /root/имя_программы, в рабочую /root. Настройки из других директорий так же легко копируются.

frugal (экономная или безопасная) установка — козырь puppy Linux. Такой простой установки нет ни в одном дистрибутиве. Тем более, что сейчас можно использовать до 25 sfs (в puppy 4.1) Как показывает опыт, такая установка очень устойчива к сбою электричества или случайному выключению.

В случае puppy 3 (и puppyrus) надо периодически удалять **wh.файлы**, чтобы не было проблем со «слоями» (это немного отдельная тема)

О Puppy Linux

Введение. На данный момент дистрибутивы Linux выпускают с основательно проработанным интерфейсом и со всякими удобствами. В сравнении с другими дистрибутивами Puppy Linux выглядит устарелым и менее привлекательным. Puppy Linux может и не выиграет конкурс по красоте, однако, тут важно что внутри а не снаружи. Если вы взгляните на дистрибутив не обращая внимание на внешний вид, то обнаружите жемчужину дистрибутива Linux.

Puppy Linux написан австралийским профессором Барри Каулером (Barry Kauler). Дистрибутив создан, чтобы быть малым, эффективным и дружественным к пользователю. К этой категории относятся хорошо знакомые дистрибутивы, такие как Damn Small Linux, SLAX и SAM Linux, но у Puppy Linux есть серьезные преимущества:

- Собран практически с нуля. Puppy очень мал и не требователен к железу.
- При загрузке с CD весь дистрибутив загружается в оперативную память и стартует без нужды доступа на CD, что делает Puppy очень быстрым.
- Puppy дает возможность сохранять данные сессии в отдельный файл, даже если вы запускаете дистрибутив с CD-RW.

- Puppy Linux устанавливается на любые носители включая USB флешь, жесткий диск или на карту памяти.
- Системную конфигурацию можно легко изменить при помощи удобного инструмента настройки.
- Puppy Linux включает в себя быстрые и удобные приложения для интернета, офиса, графики, видео, звука и даже несколько игр для развлечения.
- Puppy включает в себя собственный файловый менеджер, делая установку дополнительных приложений простой.

В результате делает Puppy Linux идеальным дистрибутивом для использования на старых компьютерах.

Так же как и с любым другим Linux дистрибутивом для начала вам нужно скачать ISO образ Puppy последней версии и записать его на CD. Убедитесь, что в БИОСе первичное загрузочное устройство назначен CD привод.

Подобно другим Live CD дистрам, Puppy поддерживает загрузочные параметры. Например, puppy pfix = ram параметр заставляет Puppy Linux загружаться в RAM без загрузки сохраненной сессии, в тоже время puppy pfix = purge делает глобальную зачистку файлов, которая может быть очень полезна для восстановления системы. Полный лист загрузочных параметров и их дескрипторы можно посмотреть на странице WIKI Puppy Linux.

В процессе загрузки вы должны выбрать графический сервер X, состоящий из двух опций Xorg и Xvesa. Xorg поддерживает множество продвинутых настроек для современного железа, но может не запуститься на старых компьютерах. Xvesa имеет ограниченное количество настроек, но запускается практически на любой конфигурации компьютера. Обычно пользователи выбирают сначала Xorg, если экран после этого ничего не показывает, то можно выбрать Xvesa. Как только Puppy загрузился нужно выбрать оптимальное разрешение экрана.

У Puppy есть отличная система управлением разрешений. Все что вам нужно это выбрать нужное разрешение и нажать кнопку TEST. Если все отображается на экране корректно, но можно продолжить работу, нажав Okay. Также можно определить разрешение вручную. Как только Puppy окончательно загрузился, взгляните на картинку на рабочем столе, которая содержит в себе несколько подсказок, включая информацию о доступной RAM памяти, состоянием подключения к интернету и сохранение ваших настроек и данных.

Установка Puppy Linux. Хотя Puppy Linux отлично запускается с CD, вы также можете установить его на любой носитель. Puppy включает в себя собственный установщик. Запустите его, выбрав в меню Menu → Setup → Puppy universal installer. Установщик включает в себя подробную информацию о процессе установки, и мы рекомендуем, чтобы вы прочли все внимательно, выбирая нужные опции.

Например, для загрузки puppy с USB флешь, которая использует файловую систему FAT32, вам нужно установить файлы в загрузочный сектор. Для установки файлов в загрузочный сектор выберете опцию mbr.bin когда появится диалог со списком доступных загрузчиков. Если вы устанавливаете Puppy в новую USB флешку, то скорее всего она не отформатирована как загрузочное устройство. В этом случае вы должны запустить GParted.

Запустите GParted, нажмите на разделе флешки правой кнопкой мыши и выберете Manage Flags. Далее выбираем Boot и жмем ОК для закрытия окна и подтверждаем наши изменения, нажав кнопку Apply. Затем закрываем GParted и установщик доделает все остальное самостоятельно.

Установить Puppy на жесткий диск также легко. Вам нужно выбрать между минимальной (frugal) и полной (full) установкой. При минимальной установки Puppy просто скопирует несколько файлов (vmlinuz, initrd.gz, pup_301.sfs и zdrv_301.sfs) с CD на выбранный логический диск, что позволяет вам запускать Puppy Linux как Live CD дистро, только с жесткого диска и сохраняя сессию и данные на жестком диске. Так же вам необходимо настроить загрузчик GRUB вручную. Полная установка позволяет установить весь дистрибутив на жесткий диск в выбранный вами логический диск.

Запуск Puppy Linux с QEMU. Puppy, установленный на USB флешь, делает дистрибутив очень компактным. Вместо того, чтобы таскать с собой ноутбук, вы можете с помощью флешки запустить Puppy на любом компьютере. Однако, в некоторых случаях вам не разрешат перезагрузить Windows и зайти в Puppy Linux. QEMU Manager - это эмулятор, позволяющий запустить Puppy на платформе Windows. Также немаловажно, что QEMU Manager - это компактная программа и поэтому ее можете установить на USB флешь с Puppy Linux. Чтобы создать виртуальную машину, основанную на QEMU с Puppy Linux, нужно скачать программу QEMU и образ последней версии Puppy Linux. Распаковать QEMU Manager и скопировать папку на USB флешь. Копируем ISO образ в каталог с QEMU Manager и запускаем QemuManager.exe. Нажимаем Create New Virtual Machine, далее появится помощник, который поможет настроить новую виртуальную машину. Все опции помощника вполне понятные и вы без проблем сможете установить новую виртуальную машину (ВМ).

Как только все шаги в создании ВМ пройдены, проверьте, что отмечен пункт View Advanced Configuration Options After Saving Box. Далее жмем кнопку Save Virtual Machine, которая сохраняет новую ВМ и открывает окно настроек. Далее переходим на вкладку Disk Configuration. В секции CD-ROM жмем кнопку Browse и выбираем ISO образ с Puppy Linux. Выбираем опцию Boot From CD-ROM. Сохраняем настройки, нажав на кнопку Save и теперь, вы можете закрыть окно. После этого можно запустить Puppy на ВМ нажав на кнопку Launch.

Настройка Puppy Linux. Puppy Linux имеет панель управления, которая позволит вам без проблем настроить ОС. Чтобы вызвать панель управления, выбираем Menu → Setup → Wizard

Wizard. Эта панель управления поможет вам настроить любой аспект Puppy, включая локальные настройки, звук, X видео, соединение с интернетом и файрвол. Если Puppy не настроил, как следует драйвера на WIFI, вы можете установить их вручную. Для этого нажмите Load Module, выберете нужный модуль из списка драйверов, и жмем Load. Если драйвер для вашей WIFI карты нету в списке, у вас есть возможность установить драйвер для Windows при помощи NDISwrapper. Перейдите в секцию More, выбираем NDISwrapper, указываем нужный драйвер и жмем OK.

Как только файл загружен, вам нужно создать новый профиль (New profile). Указываем нужное устройство, жмем кнопку appoptiate, выбираем Wireless, Create new profile и заполняем требуемые поля. Помощник поддерживает мультипрофили. С помощью него можно переключаться между различными беспроводными сетями. Для возвращения настроек по-умолчанию можно воспользоваться утилитой Menu → Desktop → Puppybackground image. Также можете удалить иконку с рабочего стола. Жмем правой кнопкой мыши на нужной иконке и выбираем Remove. Если вы выбрали минимальную установку или вы запускаете Puppy с USB флешки или другого съемного устройства все ваши настройки и данные сохранятся в отдельном файле pup_save.2fs. При следующей загрузке Puppy автоматически загрузит созданный pup_save.2fs файл.

Установка приложения. Puppy Linux имеет свой собственный менеджер пакетов, который можно использовать для установки дополнительных пакетов с официального репозитария. Puppy использует свой собственный формат называемый PET, поэтому список приложений доступных в PET пакетах не большой, но он содержит основные приложения такие как Mozila FireFox, OpenOffice.org, GIMP и другие. Для установки приложения при помощи Puppy package manager просто, нужно всего лишь выделить нужное приложение и нажать Okay. Затем Manager скачивает выбранный пакет, проверяет его целостность и устанавливает. Кроме того можно управлять и .deb пакетами, которые позволяют вам пользоваться Дебиановкими пакетами. Для доступа к этой функции вам нужно установить 2 пакета при помощи Puppy Package Manager: Веб браузер Dillo и pb_debiainstaller.

Далее можно скачивать .deb пакеты с Дебиановкого репозитария. Запускаем Терминал Menu → Utility → RXVT Terminal Emulator и вводим команду pb-debianinstaller. Эта команда запустит установщик и браузере Dillo. Наживаем кнопку Choose и выбираем скачанный .deb пакет, жмем Check dependencies и устанавливаем требуемые пакеты, если таковы требуются. После нажатия кнопки Install now и Finish, все готово.

После этого можете запустить установленную программу из терминала. Для удаления установленной программы вы можете использовать Puppy package manager. После установки пакетов Debian имейте ввиду что pb_debianinstaller все еще экспериментальная версия, и может сделать вашу систему нестабильной. Используйте эту программу с осторожностью и не забывайте делать резервную копию системы.

Сборка дистрибутива Puppy Linux. После того, как вы настроили систему и установили нужные приложения, вы можете собрать свой собственный дистрибутив Puppy Linux. Нужная программа включена в Puppy (Menu → Setup → Remaster Puppy Live-CD) позволяет вам пересобрать его всего в с помощью нескольких кликов. Программа всего-навсего создает файл pup_301.sfs (Где 301 - номер версии Puppy), создает ISO образ и записывает его на CD-DVD. Все что вам нужно это выбрать логический или диск, из которого программа сделает ISO образ.

Заключение. Со всем вниманием и рекламой, которые окружают основные дистрибутивы, такие как Ubuntu, Mandriva и openSUSE, легко пропустить маленькое чудо как Puppy Linux. Этот дистрибутив быстро загружается, даже на старом железе, его можно установить на любой носитель. Вдобавок к этому, Puppy использует свой собственный файловый менеджер, имеет возможность устанавливать дебиан приложения, а также возможность пересобрать дистрибутив. И у вас будет удивительный Linux дистрибутив.

Лабораторная работа № 10

Тема: УСТАНОВКА 4-х ОС НА ПЭВМ

Цель: Научиться устанавливать различные операционные системы: Win-XP + 3 Linux: Alt, Mops, Puppy на ПЭВМ.

Задание:

1. Произвести разбиение винчестера ПЭВМ на разделы с помощью программы fdisk следующим образом:

sda1 — 30 Gb — ntfs для Windows

sda2 — 40Gb - a5

sda3 — 2Gb - swap

sda4 — extended

sda5 — 30 Gb - ALTLinux

sda6 — 30 Gb - MOPS

sda7 — всё что остаётся, для Puppy.

2. Установить четыре операционные системы: Windows + 3 Linux: Alt, Mops, Puppy – на ПЭВМ лаборатории.

Порядок сдачи лабораторной работы

1. Продемонстрировать работу с установленными системами.
2. Предоставить отчет, в котором должно быть:
   а) задание на работу;
   б) описание подготовки винчестера (разделов),
   в) описание установки дистрибутивов,
   г) описание установки и настройки загрузчика.

Указания к выполнению работы

Свободно распространяемые версии дистрибутивов скачать с сайтов разработчиков или др. Учебники и пособия по Linux, прежде всего, рекомендуется смотреть на сайтах:

http://www.altLinux.org , http://docs.puppyrus.org, http://uco.puppyrus.org/stati

Дополнительная справочная информация

Основные шаги по подготовке HDD к использованию

Подключение. HDD, CD-Rom'ы, DVD-Rom'ы и другие аналогичные устройства хранения информации подключаются к ПЭВМ (не сильно новых) с помощью интерфейса IDE (параллельный ATA - PATA). Интерфейс IDE реализуется на системных платах в виде спаренного (двухканального) контроллера IDE — разъёмы ide0 и ide1 (иногда на системных платах они помечены как ide1 и ide2). К каждому каналу (разъёму) с помощью специального интерфейсного кабеля может быть подключено два устройства, из которых один должен быть (обязательно) master'ом, а другой (обязательно) slave. Более точно используются следующие интерфейсы:

- либо интерфейс UDMA-33 (40-pin'овый кабель) — в этом случае необходимо вручную с помощью перемычек на HDD установить статус (режим работы HDD): master или slave;

- либо интерфейс UDMA-66/100/133 (80-pin'овый кабель) — в этом случае кто есть кто определяется кабелем (цветной разъём — на разъём системной платы, серый — slave, чёрный — мастер), а перемычки на HDD устанавливаются в положение cs — cabel select.

На более новых ПЭВМ вместе с параллельнам ATA используется также последовательный ATA (serial ATA — SATA). В этой разновидности интерфейса на один разъём на системной плате предусмотрено подключение только одного устройства и потому перемычки на HDD не предусмотрены. То есть, с помощью каждого кабеля SATA подключается только один HDD и он всегда master.

Разбиение HDD на разделы. Разбиение HDD на разделы осуществляется с помощью программы **fdisk.** Назначение программы **fdisk** — создание и/или редактирование таблиц разделов (Partition Table — PT), основной и дополнительных.

Разделы являются контейнерами всего своего содержимого. Этим содержимым является, как правило, файловая система. Под файловой системой с точки зрения диска понимается «система разметки секторов и блоков диска для хранения файлов». После того, как на разделе создана файловая система и в ней размещены файлы операционной системы, раздел может стать загружаемым. Загружаемый раздел имеет в своих первых секторах небольшую программу, которая производит загрузку операционной системы — вторичный загрузчик. То есть, для загрузки операционной системы нужно явно запустить ее загрузчик из первых секторов раздела.

!!!Разметка диска на разделы еще не означает создания файловых систем. То есть, разбитый на разделы диск ещё не готов для использования.

В самом начале HDD находится «системная область» размером 63 сектора. Почему 63? Это «тяжёлая наследственность», оставшаяся от тех стародавних времён, когда дорожки HDD разбивались на 63 сектора и вся крайняя (наружняя, самая первая) дорожка была системной.

Самый первый сектор системной области (именно первый, с порядковым номером один; вообще-то, счёт, как обычно, идёт с нуля, но нулевой сектор на дорожке служит меткой начала дорожки) содержит Главную Загрузочную Запись (Master Boot Record — MBR) размером 446 байт — первичный загрузчик. Но! Первичный загрузчик находится в самом первом секторе системной области только в том случае, если на этом HDD установлена операционная система и при её установке было предписано установить загрузчик в MBR (то есть, в системную область диска). Windows никогда об этом не спрашивает и всегда MBR перезаписывает при установке. Unix'овые системы практически всегда спрашивают, куда ставить загрузчик. Это значит, что даже если на HDD есть ОС, то это ещё не означает, что первый сектор HDD содержит MBR, поскольку пользователь/админ может «захотеть» и, не понимая что делает, поставить первичный загрузчик в другое место, например, в какой-нибудь раздел, или даже на флешку. А, возможно, именно так и надо.

В следующих 64 байтах этого сектора содержится Главная Таблица Разделов (Partition Table — PT). Это таблица из 4-х строк, каждая строка которой содержит следующую структуру:

```
 struct pt_struct
{
        u8 bootable;        // флаг активности раздела          1 байт
        u8 start_part[3];   // координаты начала раздела 3 байта
        u8 type_part;       // системный идентификатор 1 байт
        u8 end_part[3];     // координаты конца раздела  3 байта
        u32 sect_before;  // число секторов перед разделом 4 байта
        u32 sect_total;     // число секторов в разделе     4 байта
 };
```

А последние два байта сектора MBR должны содержать число 0xAA55. По наличию этой сигнатуры BIOS проверяет, что первый блок был загружен успешно и что он действительно содержит MBR + PT.

Каждая строка таблицы PT описывает один раздел диска. Эти разделы (описанные в Главной PT) называются первичными или основными — primery. И поскольку строк четыре, то, следовательно, на HDD могут быть только четыре первичных раздела: в Linux'овом именовании это, например, hda1, hda2, hda3 и hda4.

Один из этих первичных разделов (любой, кроме первого) может быть расширенным разделом — extended. С номера 5 начинают считаться логические разделы, которые создаются **в расширенном разделе, как в контейнере**. В начале каждого логического раздела - в первый сектор раздела, пишется структура, похожая на сектор MBR с одним отличием: самого MBR в нём

нет, то есть первые 446 байт заполнены нулями. Но PT и сигнатура 0xAA55 в нём присутствуют.

Важно! Не путайте диск (винчестер, всё устройство в целом) с разделом на диске. Раздел — это часть диска.

Программа **fdisk** умеет работать с Главной PT - создавать первичные разделы, определять один из первичных разделов расширенным, создавать в расширенном разделе логические разделы. При создании логического раздела fdisk автоматически создаёт PT (вторичную) в первом секторе логического раздела.

При создании раздела пользователь должен ввести информацию о расположении и размере раздела. Эта информация используется для заполнения строки PT, описывающей этот раздел (см. рис.1).

Таким образом, в результате работы программы **fdisk** на HDD пишутся:

- нулевой сектор диска — сектор MBR с заполненной главной PT;

- и, если на HDD создаются логические разделы, то в начале каждого раздела по адресу сектора, с которого начинается раздел (самый первый сектор раздела) пишется сектор с вторичной PT, в первой строке которой описан этот логический раздел.

Создание файловой системы на разделе. Создание файловой системы на разделе— это операция форматирования раздела с помощью команды mkfs (см. лекции или man mkfs).

Монтирование раздела. Монтирование раздела для использования осуществляется командой mount (см. лекции или man mount).

Установка 4-х ОС на hdd

Варианты установки. Возможны два варианта установки:

а) у вас новый диск или «старый», но информация на нём вам не нужна;

б) у вас уже стоит на hdd некоторая система (например, Windows), тогда вам нужно освободить часть диска под новые ОС; освобождать место всегда нужно с конца hdd, иначе нумерация разделов на hdd может оказаться не соответствующей реальному порядку расположения разделов на hdd.

Разбиение диска (формат PC BIOS). Далее загружаетесь с какого-нибудь live-CD, например, puppy. Здесь важно то, что разбиение диска настоятельно рекомендуется делать в Linux, ибо в Windows вы далеко не всегда сможете переразбить диск так как надо.

В варианте а) вы весь диск программой fdisk разбиваете (переразбиваете) на разделы.

В варианте б) вы тоже самое проделываете с освобождённым на диске местом.

Этот пункт требует умения работать с программой fdisk и вообще понимать что такое

винчестер, раздел, таблицы разделов, файловые системы и как со всем этим добром работать.

Умение работать с fdisk — обязательно!

Разбиение диска для 40-гигабайтных hdd (в лаборатории, см. рис 26):

sda1—win—7Gb,

sda2—swap—1Gb,

sda3—alt—12Gb,

sda4—extended,

sda5—MOPS—12Gb,

sda6—puppy—что_останется.

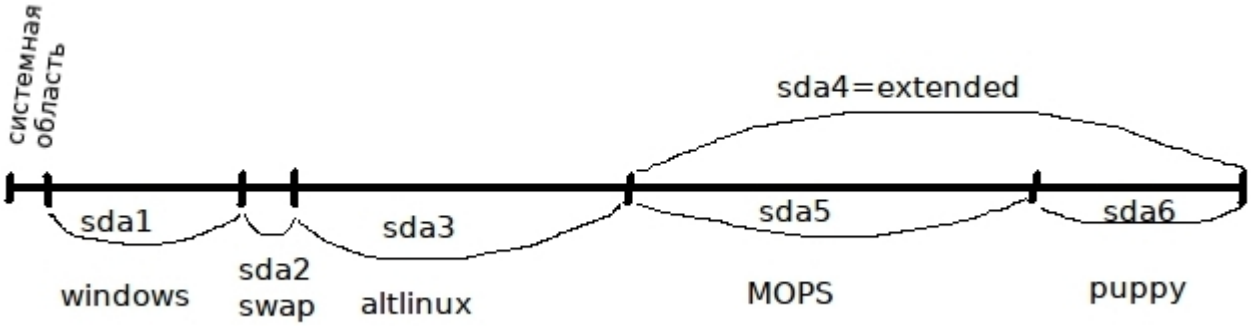


Рис. 26. Разбиение диска для 40-гигабайтных hdd

Разбиение диска для 160- или 500-гигабайтных hdd (в лаборатории, см. рис. 27) должно быть следующим:

sda1-win-30Gb,

sda2-bsd-40Gb,
sda3-swap-2Gb,

sda4-extended,

sda5-alt-30Gb,

sda6-MOPS-30Gb,

sda7-puppy-что_останется.

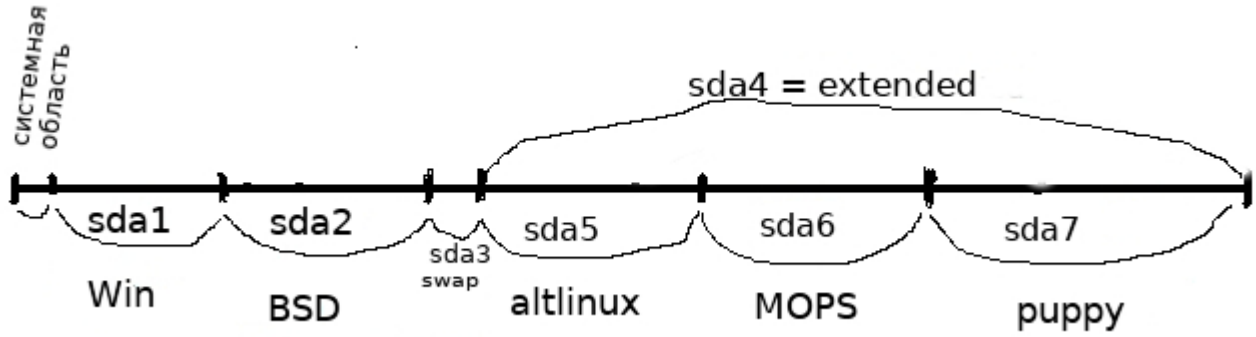


Рис. 27. Разбиение диска для 160(500)-гигабайтных hdd

В домашних условиях у вас разбиение hdd может быть другим, хотя для варианта а) вы вполне можете использовать аналогичные разбиения.

Установка операционных систем. Здесь важно соблюсти порядок установки: первой ставится Windows (поскольку она несовместима ни с чем, даже сама с собой), а затем ставятся Linux, в порядке, вам удобном.

Замечание о загрузчиках. В итоге при сдаче лабораторной вы должны продемонстрировать загрузку 4-х ОС с помощью загрузчика grub-2, именно grub 2-ой версии, а не lilo, grub4dos или ещё чего. Среди операционных систем, которые должны быть установлены на ПЭВМ (Windows, altLinux, MOPS, puppy), загрузчик grub-2 используется в altLinux и MOPS.

Пример конфигурационного файла grub-2 приведён ниже.

```
# GRUB2
set timeout=10
set default=0
set root=(hd0,3)
insmod video
insmod vbe
insmod font
```

Добрый день,

С уважением, Чекал Е.Г.

```
loadfont /boot/grub/unifont.pf2
insmod gfxterm
set gfxmode="640x480x24;640x480"
terminal_output gfxterm
insmod png
#background_image /boot/grub/grub640.png
# End GRUB global section
# Linux bootable partition config begins
```

```
menuentry "ALTLinux 7.03 KDesktop on /dev/sda3" {
	insmod ext2
	set root=(hd0,3)
	set gfxpayload="1024x768x24;1024x768"
	Linux (hd0,3)/boot/vmlinuz root=/dev/sda3 ro acpi=force quiet splash
	initrd (hd0,3)/boot/initrd.img
}
#
menuentry "MOPSLinux 7.0 on /dev/sda5" {
	set root=(hd0,5)
	set gfxpayload="1024x768x24;1024x768"
	Linux /boot/vmlinuz ROOTDEV=6142e232-da21-431c-96ee-2e74aea986b5 ro acpi=force quiet splash
	initrd /boot/initrd.gz
}
#
menuentry "MOPSLinux 7.0 (режим восстановления) на /dev/sda5" {
	set gfxpayload="1024x768x24;1024x768"
	Linux (hd0,5)/boot/vmlinuz root=/dev/sda5 ro acpi=force quiet splash single
}
#
menuentry "PuppyLinux 2.03 KDesktop on /dev/sda6" {
#	insmod gzio
#	insmod part_msdos
	insmod ext2
	set root=(hd0,6)
	set gfxpayload="1024x768x24;1024x768"
	Linux (hd0,6)/boot/vmlinuz root=/dev/sda6 ro acpi=force quiet splash
	initrd (hd0,6)/boot/initrd.gz
}
#
# Other bootable partition config begins
menuentry "Windows XP on /dev/sda1" {
	set root=(hd0,1)
	chainloader +1
```

```
}
#
menuentry "Memtest86+-4.20" {
	insmod part_msdos
	insmod ext2
	set root='hd0,3'
	search --no-floppy --fs-uuid --set=root 0ed46199-81a7-4f32-8657-caab1f 45115c
	  Linux16 /boot/memtest-4.20.bin
}
```

Лабораторная работа № 11

Тема: ПРОГРАММИРОВАНИЕ: РАБОТА С ПРОЦЕССАМИ

Цель: Научиться разрабатывать консольные программы работы с процессами

Задание:

Разработать консольную программу на языке С, согласно приведенным вариантам.

Варианты:

Задание 1. Написать программу, которая:

- создаёт подпроцесс,
- процессы идентифицируют себя, печатая сообщения и свой PID.

Задание 2. Написать программу, которая:

- печатает на дисплее собственный текст.

Задание 3. Написать программу, которая:

- создаёт подпроцесс,
- процессы идентифицируют себя, печатая сообщения и свой PID,
- процесс-потомок печатает на дисплее собственный текст.

Задание 4. Написать программу, которая:

- создаёт файл с именем <имя программы>.txt и пишет в него 10 строк: «N строки: этот файл создан программой <имя программы>-автор <ФИО>».

Задание 5. Написать программу, которая:

- создаёт файл с именем <имя программы>.txt и пишет в него исходный текст программы; в конце, с новой строки - «Автор — ФИО».

Задание 6. Написать программу, которая:

- открывает файл с исходным текстом программы и выводит его на экран.

Задание 7. Написать программу, которая:

- создаёт новый поток исполнения,
- основной процесс и поток идентифицируют себя, печатая сообщения (своё имя) и свой PID.

Задание 8. Написать программу, которая:

- создаёт новый поток исполнения,
- основной процесс и поток идентифицируют себя, печатая сообщения (своё имя) и свой PID,
- в основном процессе производится вычисление факториала 10 и печатается результат,
- в потоке производится вычисление факториала 12 и печатается результат.

Задание 9. Написать программу, которая:

- создаёт новый поток исполнения,
- основной процесс и поток идентифицируют себя, печатая сообщения (своё имя) и свой PID,
- в потоке производится вычисление факториала 12,
- в основном процессе печатается результат вычисления, сделанного в потоке.

Задание 10. Написать программу, которая:

- может обработать аргументы (ключи) строки запуска -a, -b, -c, -d и -e;
- реакция на ключи: печать сообщения «Имя программы: задан аргумент «-ключ» - обработано»,
- если ни один ключ не задан, то выдать сообщение: «Usage: <имя программы> -a, -b, -c, -d, -e».

Задание 11. Написать программу, которая:

- может обработать аргументы (ключи) строки запуска -a, -b, -c, -d, -e;
- реакция на ключи: печать сообщения «Имя программы: задан аргумент «-ключ» - обработано»,
- если задан ключ -b, то программа выводит на экран собственный текст.,
- если ни один ключ не задан, то выдать сообщение: «Usage: <имя программы> -a, -b, -c, -d, -e».

Задание 12. Написать программу, которая:

- может обработать аргументы (ключи) строки запуска -a, -b, -c, -d, -e;
- реакция на ключи: печать сообщения «Имя программы: задан аргумент «-ключ» - обработано»,
- если задан ключ -a, то программа создаёт файл с именем <имя программы>.txt и пишет в него 10 строк: «N строки: этот файл создан программой <имя программы>-автор <ФИО>»,
- если ни один ключ не задан, то выдать сообщение: «Usage: <имя программы> -a, -b, -c, -d, -e».

Задание 13. Написать программу, которая:

- может обработать аргументы (ключи) строки запуска -a, -b, -c, -d, -e;
- реакция на ключи: печать сообщения «Имя программы: задан аргумент «-ключ» - обработано»,
- если задан ключ -c, то программа создаёт файл с именем <имя программы>.txt и пишет в него исходный текст программы; в конце, с новой строки - «Автор — ФИО»,

- если ни один ключ не задан, то выдать сообщение: «Usage: <имя программы> -a, -b, -c, -d, -e».

Задание 14. Написать программу, которая:

- может обработать аргументы (ключи) строки запуска -a, -b, -c, -d, -e;
- реакция на ключи: печать сообщения «Имя программы: задан аргумент «-ключ» - обработано»,
- если задан ключ -c, то программа создаёт файл с именем <имя программы>.txt и пишет в него исходный текст программы; в конце, с новой строки - «Автор — ФИО»,
- если ни один ключ не задан, то выдать сообщение: «Usage: <имя программы> -a, -b, -c, -d, -e».

Задание 15. Написать программу, которая может обработать аргументы (ключи) строки запуска -a, -b, -c, -d, -e. Исполняемый файл программы должен называться proga. Создать жёсткую ссылку на этот файл proga1. Работа программы:

- если программа запущена из исполняемого файла с именем proga с каким-либо ключом (ключами), то печатается сообщение «Имя программы: задан аргумент «-ключ» - обработано»,
- если программа запущена из исполняемого файла с именем proga1, то выдаётся сообщение: «Error: Неверный вызов программы.
  Usage: proga -a, -b, -c, -d, -e».

Задание 16. Написать программу, которая может обработать аргументы (ключи) строки запуска -a, -b, -c, -d, -e. Исполняемый файл программы должен называться proga. Создать жёсткую ссылку на этот файл proga1. Работа программы:

- если программа запущена из исполняемого файла с именем proga с каким-либо ключом (ключами), то печатается сообщение «Имя программы: задан аргумент «-ключ» - обработано»,
- если программа запущена из исполняемого файла с именем proga1, то программа создаёт файл с именем proga.txt и пишет в него исходный текст программы; в конце, с новой строки - «Автор — ФИО»,

Порядок сдачи лабораторной

1) Демонстрация функционирования программы в лаборатории. Доработка или исправление по указанию преподавателя.

2) Предоставить отчет, в котором должно быть:

а) задание на работу;

б) описание программы;

г) исходные тексты программы;

3) Вместе с отчётом в электронном виде сдаётся исполняемый файл и исходники.

Лабораторная работа № 12

Тема: ПРОГРАММРОВАНИЕ: УЧЕТ ПОЛЬЗОВАТЕЛЕЙ ОС

Цель: Научиться разрабатывать системные программы учета пользователей

Задание:

Разработать программу, удовлетворяющую ниже приведенным требованиям.

1. Интерфейс может быть

- терминальный режим;
- псевдографика Perl или Tcl/Tk — тоже терминальный режим;
- графический режим.

Соответственно определяются четыре варианта реализации:

- терминальный (на языке C);
- терминальный с псевдографикой (на языках Perl или Tcl/tk);
- графический-1 (на языке C++);
- графический-2 (на языке Java).

2. Языки программирования: Perl, Tcl/Tk, C, C++, Java. Использование других языков предварительно согласовать.

3. Описания структур данных pwd и utmp смотреть в манах utmp, getutent и в хидерах /usr/include/pwd.h и /usr/include/utmp.h.

4. Интерфейс системы должен обеспечивать

- получение статистики о пользователях в табличной форме на экране;
- печать отчётов (таблиц, отображаемых на экране);
- работа с системой: в терминальном режиме и режиме псевдографики — клавиатура, в графическом режиме — мышь и клавиатура.

Порядок сдачи лабораторной работы

1) Демонстрация функционирования программы в лаборатории. Доработка или исправление по указанию преподавателя.

2) Представить отчет, в котором должно быть:

а) задание на работу;

б) документ «Описание программы» и документ «Руководство оператора»;

г) исходные тексты системы;

3) Вместе с отчётом в электронном виде сдаётся исполняемый файл и исходники.

Указания к выполнению работы

Для реализации графического интерфейса рекомендуется использовать библиотеку xlib (см. dfe/petrsu.ru/posob/X/index.html) либо библиотеку QT.

ИСПОЛЬЗОВАННАЯ И РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. David Garlan and Mary Shaw, An Introduction to Software Architecture, Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.

2. Боуман И. Концептуальная архитектура Ядра Linux. 1998 г. : Русский перевод: Шевченко Д., 2006 г. - URL: http://

3. URL: //syscalls.kernelkrok.com — дата доступа 06.01.14

4. URL: //j00ru.vexillium.org/win32k_syscalls — дата доступа 06.01.14

5. URL: //top500.org — дата доступа 07.01.14

6. Bach M.J.. The design of the Unix Operating System.- Prentice-Hall, 1986

7. Tanenbaum A.S.. Modern Operating Systems. - Prentice Hall, 1992

8. Ахо В., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. "Вильямс". 2001.

9. Беляков М.И., Рабовер Ю.И., Фридман А.Л. "Мобильная операционная система", М., Радио и связь, 1991.

10. Дейтел Г. Введение в операционные системы. М.: Мир. 1987.

11. Дунаев С. Unix. System V. Release 4.2. М.: Диалог МИФИ. 1996.

12. Керниган Б. В, Пайк Р. Unix - универсальная среда программирования. М.:Финансы и статистика. 1992.

13. Олифер В.Г., Олифер Н.А.. Сетевые операционные системы. - Издательский дом <Питер>, 2001.

14. Робачевский Андрей. Операционная система Unix. - BHV, 1999.

15. Цикритис Д.,Бернстайн Ф.. Операционные системы. М.: Мир. 1977.

16. Снейдер Й. "Эффективное программирование TCP/IP", Питер, 2001

17. Stevens R. W, "Unix Network Programming", Prentice Hall, Inc., volume 1-2, 1998, Second edition.

18. Таненбаум Э., Вудхалл А. Операционные системы. Разработка и реализация. 3-е изд. — Спб.: Питер, 2007. — 704 с: ил.

19. ГОСТ 19781-90. Обеспечение систем обработки информации программное. Термины и определения. — М.:Изд-во стандартов, 1990.

Приложение 1.

Структура task_struct

Структура task_struct определена в файле include/Linux/sched.h:

```
---------------------------------------------------------------------
include/Linux/sched.h
384   struct task_struct {
385     volatile long state;
386     struct thread_info *thread_info;
387     atomic_t usage;
388     unsigned long flags;
389     unsigned long ptrace;
390
391     int lock_depth;
392
393     int prio, static_prio;
394     struct list_head run_list;
395     prio_array_t *array;
396
397     unsigned long sleep_avg;
398     long interactive_credit;
399     unsigned long long timestamp;
400     int activated;
401
302     unsigned long policy;
403     cpumask_t cpus_allowed;
404     unsigned int time_slice, first_time_slice;
405
406     struct list_head tasks;
407     struct list_head ptrace_children;
408     struct list_head ptrace_list;
409
410     struct mm_struct *mm, *active_mm;
...
413     struct Linux_binfmt *binfmt;
414     int exit_code, exit_signal;
415     int pdeath_signal;
...
419     pid_t pid;
```

```
420     pid_t tgid;
...
426     struct task_struct *real_parent;
427     struct task_struct *parent;
428     struct list_head children;
429     struct list_head sibling;
430     struct task_struct *group_leader;
...
433     struct pid_link pids[PIDTYPE_MAX];
434
435     wait_queue_head_t wait_chldexit;
436     struct completion *vfork_done;
437     int __user *set_child_tid;
438     int __user *clear_child_tid;
439
440     unsigned long rt_priority;
441     unsigned long it_real_value, it_prof_value, it_virt_value;
442     unsigned long it_real_incr, it_prof_incr, it_virt_incr;
443     struct timer_list real_timer;
444     unsigned long utime, stime, cutime, cstime;
445     unsigned long nvcsw, nivcsw, cnvcsw, cnivcsw;
446     u64 start_time;
...
450     uid_t uid,euid,suid,fsuid;
451     gid_t gid,egid,sgid,fsgid;
452     struct group_info *group_info;
453     kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
454     int keep_capabilities:1;
455     struct user_struct *user;
...
457     struct rlimit rlim[RLIM_NLIMITS];
458     unsigned short used_math;
459     char comm[16];
...
461     int link_count, total_link_count;
...
467     struct fs_struct *fs;
...
469     struct files_struct *files;
...
509     unsigned long ptrace_message;
510     siginfo_t *last_siginfo;
...
516   };
```