1. История развития концепций и архитектурных решений UNIX.

История ОС насчитывает примерно полвека. Она во многом определялась и определяется развитием элементной базы и вычислительной аппаратуры. Первые ламповые вычислительные машины, появившиеся в начале 40-х годов, работали без ОС. Прообразом современных ОС явились мониторные системы (первые ОС) середины 50-х, которые автоматизировали действия оператора по выполнению пакета заданий.

В 1965-1975 были реализованы практически все основные концепции, присущие современным ОС: мультипрограммирование, мультипроцессирование, многотерминальный режим, виртуальная память, файловые системы, разграничение доступа и сетевая работа. Реализация мультипрограммирования потребовала внесения очень важных изменений в аппаратуру компьютера. В процессорах появился привилегированный и пользовательский режимы работы, специальные регистры для быстрого переключения с одной задачи на другую, средства защиты областей памяти, а также развитая система прерываний.

В конце 60-х были начаты работы по созданию глобальной сети ARPANET, явившейся отправной точкой для Интернета, — глобальной общедоступной сети.

К середине 70-х годов широкое распространение получили мини-компьютеры, появились первые сетевые ОС. Экономичность и доступность мини-компьютеров послужила мощным стимулом для создания локальных сетей. Первые локальные сети строились с помощью нестандартного коммуникационного оборудования и нестандартного программного обеспечения.

В конце 70-х годов был создан рабочий вариант стека протоколов TCP/IP. В 1983 году стек протоколов TCP/IP был стандартизован. Появляются специализированные ОС, например, для систем реального времени. В таких ОС часто отсутствовала поддержка мультипрограммирования.

Начало 80-х годов связано с появлением персональных компьютеров, которые послужили катализатором для бурного роста локальных сетей. В результате поддержка сетевых функций стала для ОС персональных компьютеров необходимым условием.

К началу 90-х практически все ОС стали сетевыми, способными поддерживать работу с разнородными клиентами и серверами. Особое внимание в течение всего последнего десятилетия уделялось корпоративным сетевым ОС, для которых характерны высокая степень масштабируемости, поддержка сетевой работы, развитые средства обеспечения безопасности, способность работать в гетерогенной среде, наличие средств централизованного администрирования и управления.

1 января 1970 года – официальная дата рождения (на ассемблере). От нее отсчитывают время системные часы.

1972 – расширение версии 1, появление каналов, поддержка языков.

1973 – ядро и shell переписываются на C.

1987 – Таненбаум создал ОС MINIX. Первая версия UNIX для IBM PC.

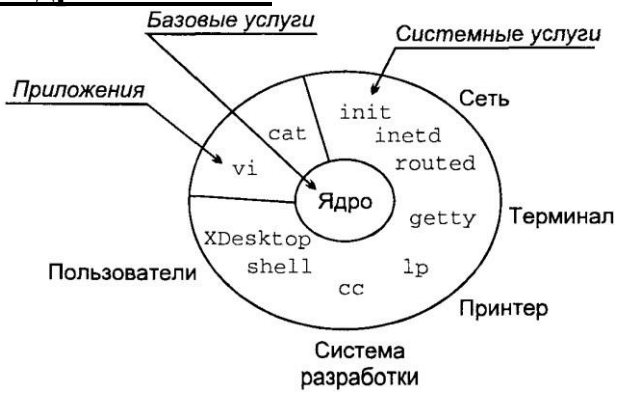
1991 – Линус Торвалдс создает ядро Linux.

Характеристики UNIX:

- Децентрализация разработки, наличие развитой системы стандартизации;
- Открытость архитектур, методов и средств;
- Стабильность основных идей и разработки на их основе новых методов и средств;
- Универсальная направленность инструментальных средств;
- Эволюционность развития архитектур UNIX-подобных ОС и переносимость приложений;
- Цивилизованные способы распространения ПО.

2. Архитектура ОС UNIX. Ядро ОС. Структура ядра ОС UNIX.

В центре находится ядро системы (kernel).
Второй уровень составляют приложения или задачи, как системные, определяющие функциональность системы, так и прикладные, обеспечивающие пользовательский интерфейс UNIX.



Структура ядра

Ядро
-
взаимодействует с аппаратной частью компьютера, изолируя прикладные программы от особенностей ее архитектуры.
- имеет набор услуг, предоставляемых программам, к ним относятся операции:
*ввода/вывода (открытия, чтения, записи и управления файлами)
*создания и управления процессами, их синхронизации и межпроцессного взаимодействия
- распределяет память
- обеспечивает доступ к файлам и периферийным устройствам.

Приложения запрашивают услуги ядра посредством *системных вызовов.* Ядро от имени процесса выполняет запрос и возвращает процессу данные
Интерфейс системных вызовов представляет собой набор услуг ядра и определяет формат запросов на услуги.

Концептуальная модель объектов управления UNIX:

- **Файловая подсистема:**

-обеспечивает унифицированный интерфейс доступа к данным (одни и те же функции могут использоваться как при чтении или записи данных на диск, так и при выводе текста на принтер);
- контролирует права доступа к файлам, которые определяют привилегии пользователя);
- размещает и удаляет файлы, выполняет запись/чтение данных файла.
- перенаправляет запросы, адресованные периферийным устройствам, соответствующим модулям подсистемы ввода/вывода.

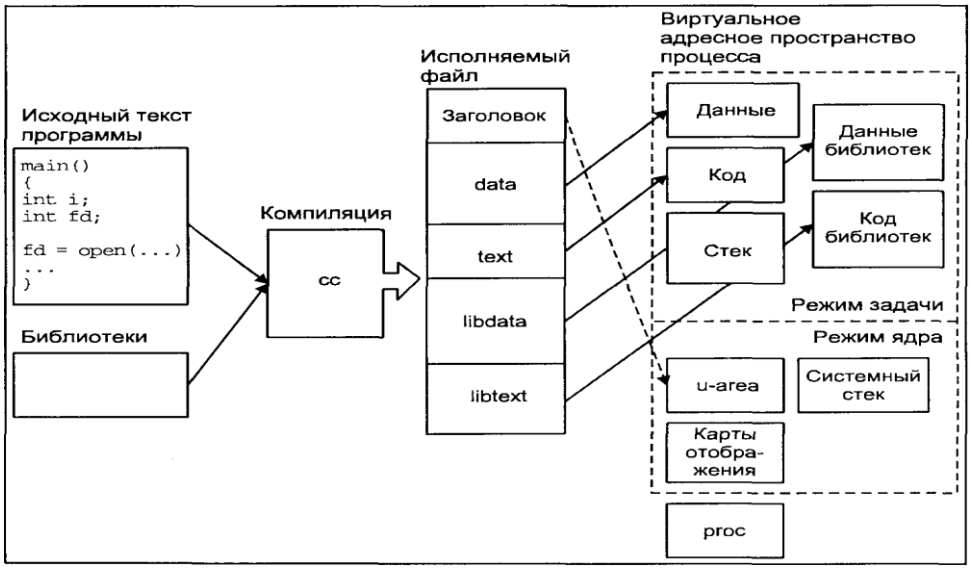
- **Подсистема управления процессами:**

- создание и удаление процессов
- распределение системных ресурсов между процессами (специальная задача ядра, планировщик, следит за тем, чтобы процесс монопольно не захватил разделяемые системные ресурсы. Процесс освобождает процессор, ожидая длительной операции ввода/вывода, или по прошествии кванта времени, после чего планировщик выбирает следующий с наивысшим приоритетом и запускает его)
- синхронизация процессов
-межпроцессное взаимодействие (соответствующий модуль уведомляет процесс о событиях с помощью сигналов и обеспечивает возможность передачи данных между различными процессами)

- **Подсистема ввода/вывода:**

-выполняет запросы файловой подсистемы и подсистемы управления процессами для доступа к периферийным устройствам, посредством взаимодействия с драйверами устройств;
- буферизация данных

3. Информационная структура процесса. Граф состояний процесса. Процесс зомби.



Процесс - представляет собой исполняемый образ программы, включающий отображение в памяти исполняемого файла, полученного в результате компиляции, стек, код и данные библиотек, а также ряд структур данных ядра, необходимых для управления процессом.

Выполнение процесса может происходить в двух режимах — **в *режиме ядра* или в *режиме задачи***. В режиме задачи процесс выполняет инструкции прикладной программы, допустимые на непривилегированном уровне защиты процессора.

Когда процессу требуется получение каких-либо услуг ядра, он делает системный вызов, который выполняет инструкции ядра, находящиеся на привилегированном уровне. Выполнение процесса при этом переходит в режим ядра. Таким образом ядро системы защищает собственное адресное пространство от доступа прикладного процесса, который может нарушить целостность структур данных ядра и привести к разрушению ОС.

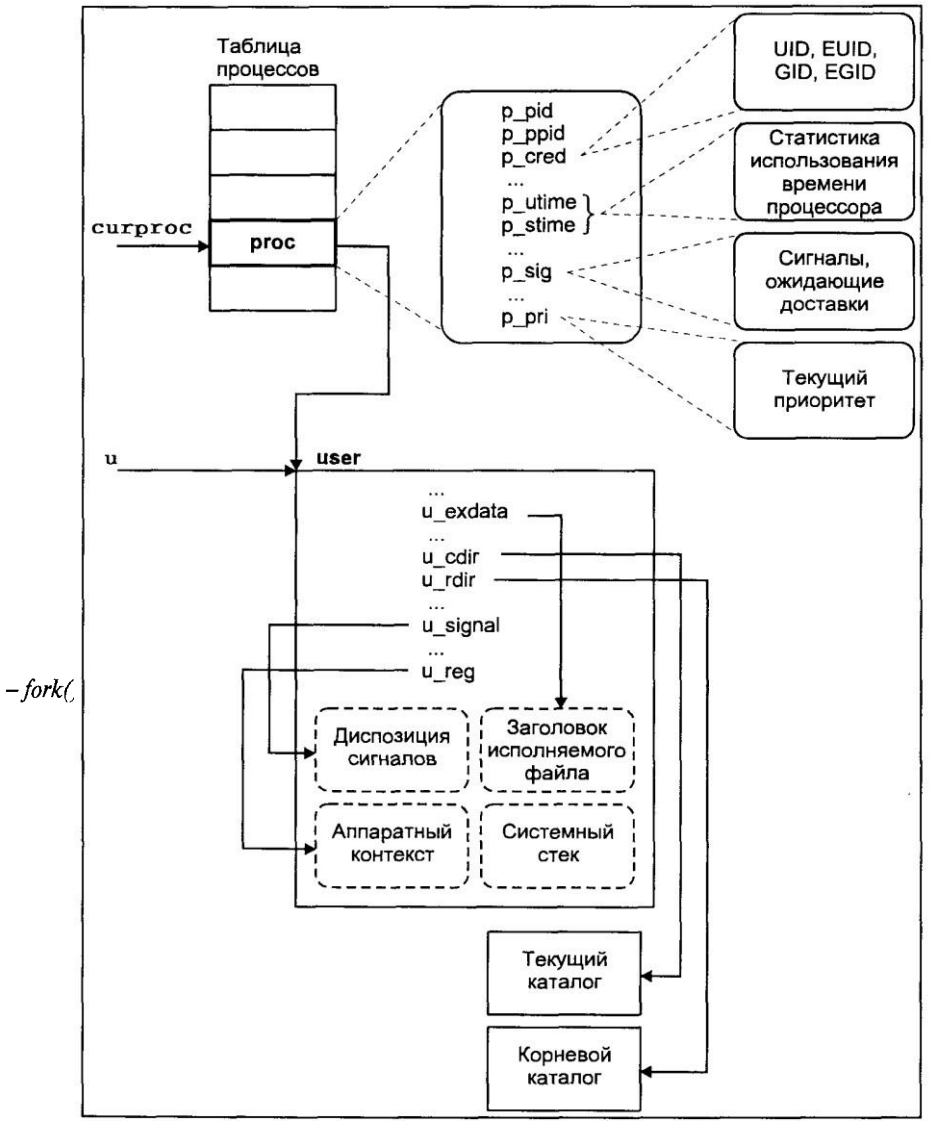
Соответственно и образ процесса состоит из двух частей: ***данных режима ядра и режима задачи***. Образ процесса в режиме задачи состоит из структур данных, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, недоступных процессу в режиме задачи, которые используются ядром для управления процессом (состояния регистров, таблицы для отображения памяти и т. д.).

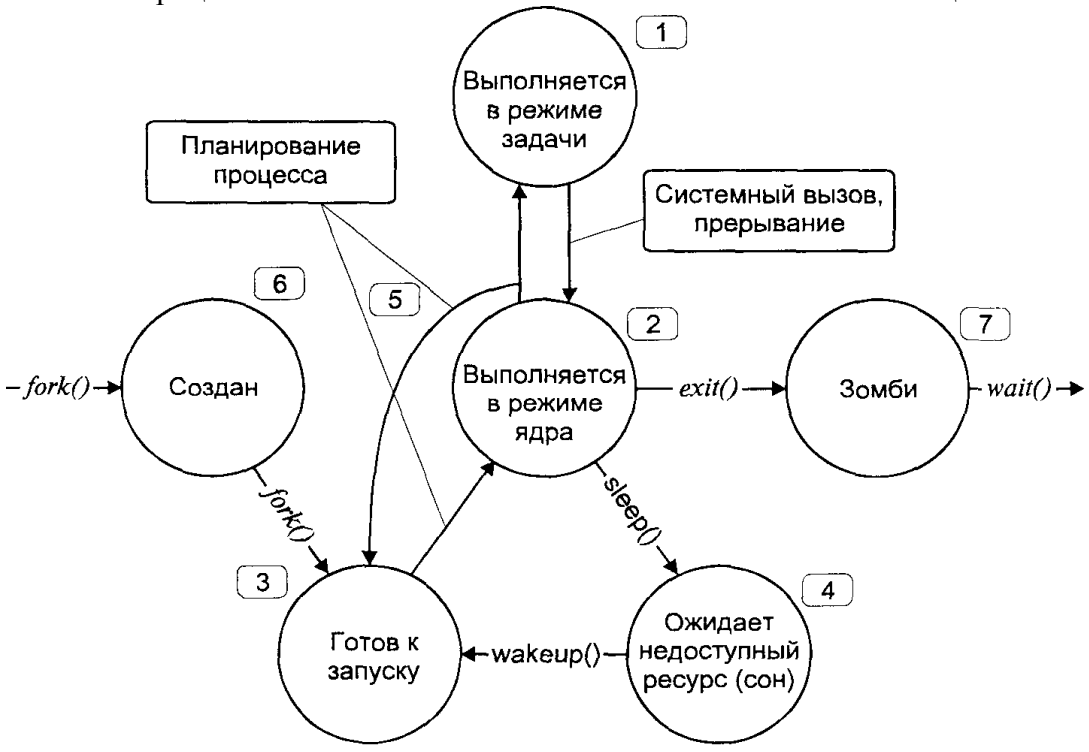
Каждый процесс представлен в системе двумя основными структурами данных — **proc и user.** Структура **proc** является записью системной таблицы процессов, которая всегда находится в ОП. Запись этой таблицы для выполняющегося в настоящий момент процесса адресуется системной переменной curproc. Это позволяет ядру иметь под рукой минимальную информацию, необходимую для определения местонахождения остальных данных, относящихся к процессу, даже если они отсутствуют в памяти.

Вторая структура — **user** (u-area) содержит дополнительные данные о процессе, которые требуются ядру только во время выполнения процесса. Данные user отображаются в определенном месте виртуальной памяти ядра и адресуются переменной u. Эти данные используются многими подсистемами ядра и не только для управления процессом (инф-ия об открытых файловых дескрипторах, диспозиция сигналов, статистика выполнения процесса, а также сохраненные значения регистров, когда выполнение процесса приостановлено). Процесс не должен иметь возможности модифицировать эти данные произвольным образом, поэтому u-area защищена от доступа в режиме задачи.

Состояния процесса:

1. *Процесс выполняется в режиме задачи*. При этом процессором выполняются прикладные инструкции данного процесса.
2. *Процесс выполняется в режиме ядра*. При этом процессором выполняются системные инструкции ядра ОС от имени процесса.
3. Процесс не выполняется, но *готов к запуску*, как только планировщик выберет его (состояние runnable). Процесс находится в очереди на выполнение и обладает всеми необходимыми ему ресурсами, кроме вычислительных.
4. Процесс находится *в состоянии сна* (asleep), ожидая недоступного в данный момент ресурса, например, завершения операции ввода/вывода.
5. *Процесс возвращается из режима ядра в режим задачи*, но ядро прерывает его и производит переключение контекста для запуска более высокоприоритетного процесса.
6. Процесс *только что создан* вызовом *fork(2)* и находится в переходном состоянии: он существует, но не готов к запуску и не находится в состоянии сна.
7. Процесс *выполнил системный вызов exit(2)* и перешел в состояние *зомби*. Как такового процесса не существует, но остаются записи, содержащие код возврата и временную статистику его выполнения, доступную для родительского процесса. Это состояние является конечным в жизненном цикле процесса.





4. Исполнимый файл и его формат. Основные форматы исполнимых файлов.

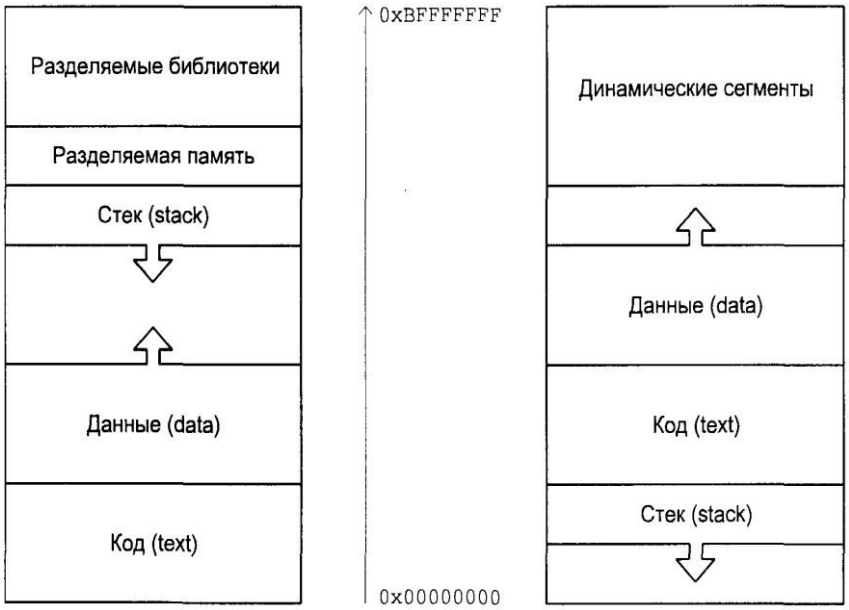
Исполнимый файл — файл, содержащий программу в виде, в котором она может быть (после загрузки в память и настройки по месту) исполнена компьютером.

Форматы исполняемых файлов

Размер, содержимое и расположение сегментов в памяти определяется как самой программой, например, использованием библиотек, размером кода и данных, так и форматом исполняемого файла этой программы. Информация, хранящаяся в исполняемых файлах форматов COFF и ELF, позволяет ответить на ряд важных вопросов:

- Какие части программы необходимо загрузить в память?
- Как создается область для неинициализированных данных?
- Какие части процесса должны быть сохранены в дисковой области свопинга?
- Где в памяти располагаются инструкции и данные программы?
- Какие библиотеки необходимы для выполнения программы?
- Как связаны исполняемый файл на диске, образ программы в памяти и дисковая область свопинга?

Формат a.out(assembler output): первоначальный формат исполняемых файлов ОС UNIX практически вытесненном форматами COFF и ELF. Простой прозрачный формат соответствовал простому устройству PDP11, но при переносе на другие системы становился «тесноват» для более сложной архитектуры.



Размер стека автоматически изменяется операционной системой, в то время как управление размером сегмента данных производится самим приложением.

Сегмент данных включает инициализированные данные, копируемые в память из соответствующих разделов исполняемого файла, и неинициализированные данные, которые заполняются нулями перед началом выполнения процесса. Неинициализированные данные часто называют сегментом BSS.

Формат ELF различает следующие типы файлов:

1. ***Перемещаемый файл,*** хранящий инструкции и данные, которые могут быть связаны с другими объектными файлами. Результатом такого связывания может быть исполняемый файл или разделяемый объектный файл.
2. ***Разделяемый объектный файл*** также содержит инструкции и данные, но:
   а) он может быть связан с другими перемещаемыми файлами и разделяемыми объектными файлами, в результате будет создан новый объектный файл.
   б) Во втором случае, при запуске программы на выполнение ОС может динамически связать его с исполняемым файлом программы, в результате чего будет создан исполняемый образ программы. В последнем случае речь идет о разделяемых библиотеках.
3. ***Исполняемый файл*** хранит полное описание, позволяющее системе создать образ процесса. Он содержит инструкции, данные, описание необходимых разделяемых объектных файлов, а также необходимую символьную и отладочную информацию.

Информация, содержащаяся в **таблице заголовков программы**, указывает ядру, как создать образ процесса из сегментов. Большинство сегментов *программы* копируются (отображаются) в память и представляют собой соответствующие сегменты процесса при его выполнении, например, сегменты кода или данных.

Каждый **заголовок сегмента** программы описывает один:

- Тип сегмента и действия ОС с данным сегментом
- Расположение сегмента в файле
- Стартовый адрес сегмента в виртуальной памяти процесса
- Размер сегмента в файле
- Размер сегмента в памяти
- Флаги доступа к сегменту (запись, чтение, выполнение)

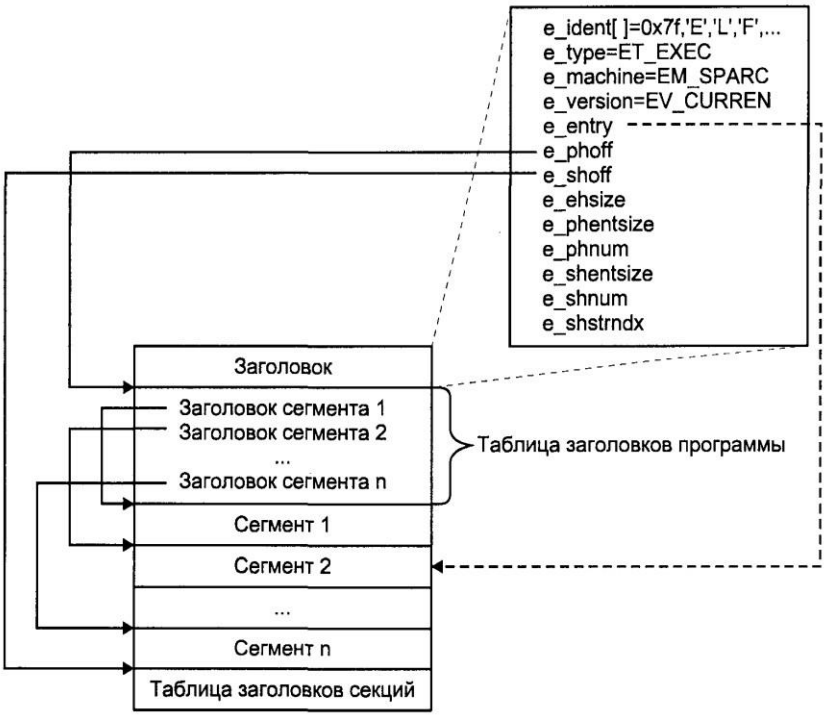
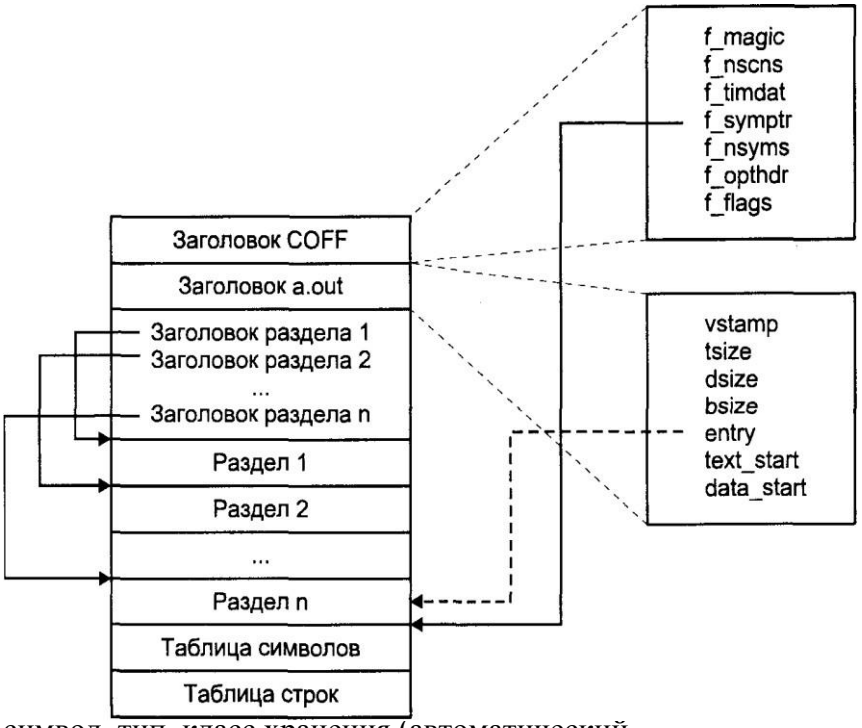


Таблица заголовков секции определяет разделы файла, используемые для связывания с другими модулями в процессе компиляции или при динамическом связывании. Соответственно, заголовки содержат всю необходимую информацию для описания этих разделов.

Формат COFF:

Все файлы формата COFF имеют один или более разделов, каждый из которых описывается своим заголовком. В заголовке хранится имя раздела (.text, .data, .bss или любое другое, установленное соответствующей директивой ассемблера), размер раздела, его расположение в файле и виртуальный адрес после запуска программы на выполнение. Заголовки разделов следуют сразу за заголовком файла.



Таблицы символов и строк являются основой системы отладки. *Символом* является любая переменная, имя функции или метка, определенные в программе.

Каждая запись в таблице символов хранит имя символа, его виртуальный адрес, номер раздела, в котором определен символ, тип, класс хранения (автоматический, регистровый и т. д.). Если имя символа занимает больше восьми байт, то оно хранится в таблице строк. В этом случае в поле имени символа указывается смещение имени символа в таблице строк. С помощью символьной информации можно определить виртуальный адрес некоторого символа. Одним из очевидных применений этой возможности является использование символьной информации в программах-отладчиках. Эта возможность используется некоторыми программами, например, утилитой *ps(l),* отображающей состояние процессов в системе.

5. Принципы планирования процессов. Контекст процесса. Переключение контекста.

Планировщик процессов обеспечивает предоставление процессорных ресурсов процессам, выполняющимся в ОС. Каждому процессу вычислительные ресурсы выделяются на ограниченный промежуток времени, после чего они предоставляются другому процессу и т. д.
Максим. временной интервал, на который процесс может захватить процессор, называется ***временным квантом***. Таким образом одновременно выполняются несколько приложений. Можно выделить три основных класса приложений:

- ***Интерактивные приложения*** большую часть времени обычно проводят в ожидании пользовательского ввода (нажатия клавиш клавиатуры).
- ***Фоновые приложения*** не требуют вмешательства пользователя (компиляция ПО). Для них важно минимизировать суммарное время выполнения в системе.
- ***Приложения реального времени*** требуют дополнительных системных возможностей, в частности, гарантированного времени совершения той или иной операции, времени отклика и т.п.

Обработка прерываний таймера:
Каждый комп имеет аппаратный таймер или системные часы, которые генерируют аппаратное прерывание через фиксированные интервалы времени. Временной интервал м/у соседними прерываниями называется ***тиком процессора***. Обработчик прерываний ядра вызывается аппаратным прерыванием таймера, приоритет которого обычно самый высокий. Обработчик решает следующие задачи:

- Обновление статистики использования процессора для текущего процесса
- Выполнение ряда функций, связанных с планированием процессов (пересчет приоритетов и проверку истечения временного кванта для процесса)
- Проверка превышения процессорной квоты для данного процесса и отправка этому процессу сигнала в случае превышения
- Обновление системного времени (времени дня) и других связанных с ним таймеров
- Обработка отложенных вызовов
- Обработка (alarm)
- Пробуждение в случае необходимости системных процессов

Большинство систем вводят нотацию ***главного тика***, который происходит каждые n тиков, где n зависит от конкретной версии системы. Определенный набор функций выполняется только на главных тиках.

Отложенный вызов определяет функцию, вызов которой будет произведен ядром системы через некоторое время. Функции отложенных вызовов выполняются в системном контексте, а не в контексте прерывания. Вызов этих функций выполняется отдельным обработчиком отложенных вызовов, который запускается после завершения обработки прерывания таймера. Обработчик отложенных вызовов проверяет флаги и запускает необходимые функции в системном контексте. Эти функции хранятся в системной таблице отложенных вызовов, организация которой отличается для различных версий UNIX.

Алармы. Процесс может запросить ядро отправить сигнал по прошествии определенного интервала времени. Существуют 3 типа алармов. С каждым из этих типов связан таймер интервал. Значение уменьшается на единицу при каждом тике.
ITIMER_REAL (реального времени) используется для отсчета реального времени. Когда значение таймера становится равным нулю, процессу отправляется сигнал **SIGALRM**
ITIMER_PROF (профилирования) уменьшается только когда процесс выполняется в режиме ядра или задачи. Когда значение таймера становится равным нулю, процессу отправляется сигнал **SIGPROF**
ITIMER_VIRT (виртуального времени) уменьшается только когда процесс выполняется в режиме задачи. Когда значение таймера становится равным нулю, процессу отправляется сигнал **SIGVTALRM.**
Для установки таймеров всех трех типов используется системный вызов settimer или alarm.

Контекст процесса
Каждый процесс UNIX имеет *контекст,* под которым понимается вся инф-ия, требуемая для описания процесса. Эта инф-ия сохраняется, когда выполнение процесса приостанавливается, и восстанавливается, когда планировщик предоставляет процессу выч. ресурсы. Контекст процесса состоит из нескольких частей:
Адресное пространство процесса в режиме задачи (код, данные и стек процесса и т.д.).
Управляющая инф-ия. Ядро использует 2 основные стр-ры данных для управления процессом: proc и user. Сюда же входят данные, необходимые для отображения вирт. адресного пр-ва процесса в физ.
Окружение процесса. Переменные окружения процесса - строки пар вида «переменная=значение», которые наследуются дочерним процессом от родительского и обычно хранятся в нижней части стека.
Аппаратный контекст. Значения общих и ряда системных регистров процессора:

- указатель инструкций, содержащий адрес след. инструкции, которую необходимо выполнить;
- указатель стека, содержащий адрес последнего элемента стека;
- регистры плавающей точки;
- регистры управления памятью, отвечающие за трансляцию вирт. адреса процесса в физический.

Переключение м/у процессами, необходимое для справедливого распределения вычислительного ресурса, выражается в ***переключении контекста,*** когда контекст выполнявшегося процесса запоминается и восстанавливается контекст процесса, выбранного планировщиком.
Переключение контекста явл-ся достаточно ресурсоемкой операцией. Помимо сохранения состояния регистров процесса, ядро вынуждено выполнить мн-во других действий.
Ситуации, при которых производится переключение контекста:
1. Текущий процесс переходит в состояние сна, ожидая недоступного ресурса.
2. Текущий процесс завершает свое выполнение.
3. После пересчета приоритетов в очереди на выполнение находится более высокоприоритетный процесс.
4. Происходит пробуждение более высокоприоритетного процесса.

Планирование процессов:
Основано на *приоритете* процесса. Планировщик всегда выбирает процесс с наивысшим приоритетом и может приостановить выполнение текущего процесса (с более низким приоритетом) даже если последний не «выработал» свой временной квант.
Каждый процесс имеет 2 атрибута приоритета: **текущий приоритет** на основании которого происходит планирование, и **относительный приоритет**, называемый nice number (nice), который задается при порождении процесса и влияет на текущий приоритет.
Текущий приоритет варьируется в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет).
Процессы, в режиме задачи, имеют более низкий приоритет, чем в режиме ядра. Для режима задачи приоритет меняется в диапазоне 0-65, для режима ядра — 66-95 (системный диапазон).
Процессы, приоритеты которых лежат в диапазоне 96-127 являются процессами с фиксированным приоритетом, не изменяемым ОС, и предназначены для поддержки приложений реального времени.

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение **приоритета сна** выбираемое ядром из диапазона системных приоритетов и связанное с событием, вызвавшее это состояние.
Когда процесс пробуждается, ядро устанавливает значение текущего приоритета процесса равным приоритету сна. Поскольку приоритет такого процесса находится в системном диапазоне и выше, чем приоритет режима задачи, вероятность предоставления процессу вычислительных ресурсов весьма велика. Такой подход позволяет, в частности, быстро завершить системный вызов, выполнение которого, в свою очередь, может блокировать некоторые системные ресурсы.
После завершения системного вызова перед возвращением в режим задачи ядро восстанавливает приоритет режима задачи, сохраненный перед выполнением системного вызова. Это может привести к понижению приоритета, что, в свою очередь, вызовет переключение контекста.

6. Порождение нового процесса. Системный вызов fork().

Новый процесс порождается с помощью системного вызова fork(). *Дочерний* процесс является точной копией процесса, выполнившего этот вызов, или *родительского* процесса.

В частности, дочерний процесс наследует такие атрибуты родителя, как:

1. идентификаторы пользователя и группы,
2. переменные окружения,
3. диспозицию сигналов и их обработчики,
4. ограничения, накладываемые на процесс,
5. текущий и корневой каталог,
6. маску создания файлов,
7. все файловые дескрипторы, включая файловые указатели,
8. управляющий терминал.

Виртуальная память дочернего процесса не отличается от образа родительского: такие же сегменты кода, данных, стека, разделяемой памяти и т.д. После возврата из вызова fork(), который происходит и в родительский и в дочерний процессы, оба начинают выполнять одну и туже инструкцию.

Различия между этими дочерним и родительским процессами:

1. дочернему процессу присваивается уникальный идентификатор PID
2. идентификаторы родительского процесса PPID у этих процессов различны
3. Дочерний процесс получает собственную копию и, в частности, собственные файловые дескрипторы, хотя он разделяет те же записи файловой таблицы.
4. дочерний процесс свободен от сигналов, ожидающих доставки
5. Временная статистика выполнения процесса в режиме ядра и задачи для дочернего процесса обнуляется.
6. Блокировки памяти и записей, установленные родительским процессом, потомком не наследуются.
7. значение, возвращаемое системным вызовом *fork(2)* различно для родителя и потомка.

Возврат из функции происходит как в родительский, так и в дочерний процесс. Возвращаемое родителю значение равно PID дочернего процесса, а дочерний получает значение, равное 0. Если возвращает -1 – ошибка (возврат происходит только в процесс, выполнивший системный вызов). В возвращаемом *fork()* значении заложен большой смысл, поскольку оно позволяет определить, кто является родителем, а кто — потомком, и соответственно разделить функциональность.

Вызов fork() выполняет следующие действия:

1. Резервирует место в области свопинга для сегмента данных и стека процесса.
2. Размещает новую запись proc в таблице процессов и присваивает процессу уникальный идентификатор PID.
3. Инициализирует структуру proc
4. Размещает карты отображения, необходимые для трансляции адреса.
5. Размещает u-area процесса и копирует ее содержимое с родительского.
6. Создает соответствующие области процесса, часть из которых совпадает с родительскими.
7. Инициализирует аппаратный контекст процесса, копируя его с родительского.
8. Устанавливает в ноль возвращаемое дочернему процессу вызовом *fork()* значение.
9. Устанавливает возвращаемое родительскому процессу вызовом *fork()* значение равным PID потомка.
10. Помечает процесс готовым к запуску и помещает его в очередь на выполнение.

7. Системный вызов exec(). Реальный и эффективный идентификаторы пользователя. Программы, использующие бит смены идентификатора пользователя

Запуск новой программы осуществляется с помощью системного вызова *exec()*. При этом создается не новый процесс, а новое адресное пространство процесса, которое загружается содержимым новой программы. После возврата из вызова *exec()* процесс продолжает выполнение кода новой программы. Исполняемый файл содержит заголовок, позволяющий ядру правильно разместить адресное пространство процесса и загрузить в него соответствующие фрагменты исполняемого файла. **Ряд действий, которые выполняет *exec()* для запуска новой программы:**

- Не порождает нового процесса
- Происходит замещение кода текущего процесса
- Анализирует содержимое файла
- Производит трансляцию имени файла. В результате возвращается индексный дескриптор, с помощью которого осуществляется доступ к файлу. При этом проверяются права доступа.
- Считывает заголовок файла и проверяет, является ли файл исполняемым.
- Если исполняемый файл имеет атрибуты SUID или SGID, *exec()* соответствующим образом изменяет эффективные идентификаторы UID и GID для этого процесса.
- Сохраняет аргументы вызова *exec()* и переменные окружения в адресном пространстве ядра, т.к. адресное пространство процесса будет уничтожено.
- Резервирует место в области свопинга для сегмента данных и стека.
- Освобождает старые области процесса и соответствующие области свопинга. Если процесс был создан вызовом *vfork(),* старое адр. пр-во возвращается родителю, в противном случае оно просто уничтожается.
- Размещает и инициализирует карты отображения для новых сегментов кода, данных и стека. Если сегмент кода является активным (какой-либо процесс уже выполняет эту программу) данная область используется совместно. Иначе область заполняется содержимым соответствующего раздела исполняемого файла или инициализируется нулями для неициниал. данных. Поскольку упр-ие памятью процесса построено на механизме страничного замещения по требованию, копирование происходит постранично и только тогда, когда процесс обращается к страницам, отсутствующим в памяти.
- Копирует сохраненные аргументы и переменные окружения в новый стек процесса.
- Устанавливает обработку всех сигналов по умолчанию, т.к. процесс теперь не имеет требуемых обработчиков. Установки для игнорируемых и заблокированных сигналов не изменяются.
- Инициализирует аппаратный контекст процесса. В частности, после этого указатель инструкций адресует точку входа новой программы.

Реальный (RID) и эффективный (EUID) идентификаторы пользователя (RGID и EGID)

Реальным идентиф. пользователя данного процесса яв-ся идентиф.пользователя, запустившего процесс. **Эффективный идентиф.** служит для определения прав доступа процесса к системным ресурсам (в первую очередь к ресурсам ФС). Обычно реальный и эффективный идентиф. эквивалентны, т.е. процесс имеет в системе те же права, что и пользователь, запустивший его. Но существует возможность задать процессу более широкие права, чем права пользователя путем установки флага SUID, когда эффективному идентификатору присваивается значение идентификатора владельца исполняемого файла (например, администратора).

4 идентиф., играющие решающую роль при доступе к системным ресурсам: идентиф.пользователя UID, эффективный идентиф. пользователя EUID, идентиф. группы GID и эффективный идентиф. группы EGID. Эти идентиф. определяют права процесса в ФС, и как следствие, в ОС в целом. Все процессы, которые запускаются, имеют идентиф. пользователя и идентиф. группы. Исключение составляют процессы с установленными флагами SUID и SGID.

Атрибуты (флаги) **SUID и SGID** позволяют изменить права пользователя при запуске на выполнение файла, имеющего эти атрибуты. При этом привилегии будут изменены (обычно расширены) лишь на время выполнения и только в отношении этой программы. Установка SUID приведет к наследованию прав владельца-пользователя файла, а установка SGID — владельца- группы.

Системные вызовы: int getuid(); - получить uid; int geteuid(); - получить euid; int setuid(int); - установить uid, euid, suid в одно значение; int seteuid(int); - установить euid.

Программы, использующие suid бит: su, sudo, passwd, mount, umount, ping, traceroute

8. Код завершения процесса. Системный вызов exit(). Передача параметров процессу. Окружение процесса.

Существует несколько способов завершения программы. Основными являются **возврат из функции main()** (начальная функция запуска программы на выполнение _start написана таким образом, что *exit()* вызывается автоматически при возврате из функции main(). В языке С она имеет следующий вид: exit(main(argc,argv))) и **вызов функций *exit(),*** оба приводят к завершению выполнения задачи.

Процесс также может завершиться по независящим от него обстоятельствам, например, при получении сигнала, действие по умолчанию для большинства из которых приводит к завершению выполнения. В этом случае функция *exit()* будет вызвана ядром от имени процесса.

Системный вызов *exit()* выглядит следующим образом: void exit(int status).
Аргумент status, передаваемый exit() возвращается родительскому процессу и представляет собой код возврата программы. По соглашению программа возвращает 0 в случае успеха и другую величину в случае неудачи. Значение кода неудачи может иметь дополнительную трактовку, определяемую самой программой. Наличие кода возврата позволяет программам взаимодействовать друг с другом.

Задача может зарегистрировать ***обработчики выхода*** — функции, которые вызываются после вызова *exit(),* но до окончательного завершения процесса. Эти обработчики, вызываемые по принципу LIFO (последний зарегистрированный обработчик будет вызван первым), запускаются только при "добровольном" завершении процесса. Например, при получении процессом сигнала обработчики выхода вызываться не будут.
Для обработки таких ситуаций следует использовать специальные функции — *обработчики сигналов*. Обработчики выхода регистрируются с помощью функции atexit(3C): int atexit(void (*func) (void)); Функцией *atexit(l)* может быть зарегистрировано до 32 обработчиков.

Функция exit() выполняет следующие действия:

1. Отключает все сигналы.
2. Закрывает все открытые файлы.
3. Сохраняет статистику использования вычислительных ресурсов и код возврата в записи proc таблицы процессов.
4. Изменяет состояние процесса на "зомби".
5. Делает процесс init() родительским для всех потомков данного процесса.
6. Освобождает адресное пространство процесса, u-area, карты отображения и области свопинга, связанные с процессом.
7. Отправляет сигнал SIGCHLD родительскому процессу, уведомляя его о "смерти" потомка.
8. Пробуждает родительский процесс, если тот ожидает завершения потомка.
9. Запускает функцию переключения контекста, в результате чего высокоприоритетный процесс получает доступ к вычислительным ресурсам.

После завершения выполнения функции exit() процесс находится в состоянии "зомби". При этом от процесса остается запись proc в таблице процессов, содержащая статистику использования вычислительных ресурсов и код возврата. Эта инф-ия может потребоваться родительскому процессу, поэтому освобождение структуры proc производит родитель с помощью системного вызова wait(), возвращающего статистику и код возврата потомка. Если родительский процесс заканчивает свое выполнение раньше потомка, "родительские права" переходят к процессу init(). В этом случае после смерти потомка init() делает системный вызов wait(2) и освобождает структуру proc.
Другая ситуация возникает, если потомок заканчивает свое выполнение раньше родителя, а родительский процесс не производит вызова wait(). В этом случае структура proc потомка не освобождается и процесс продолжает находиться в состоянии "зомби" до перезапуска ОС. Хотя такой процесс (которого, вообще говоря, не существует) не потребляет ресурсов системы, он занимает место в таблице процессов, тем самым уменьшая максимальное число активных задач.

Окружение процесса: Переменные окружения процесса представляют собой строки пар вида: «переменные=значение», которые наследуются дочерним процессом от родительского и обычно хранятся в нижней части стека. Обычно используются для передачи текстовой инф-ии м/у процессами.
Системные вызовы: char * getenv(const char * name) - вернуть значение переменной; int putenv(char * string) - поместить строку в окружение, параметр string станет частью окружения процесса

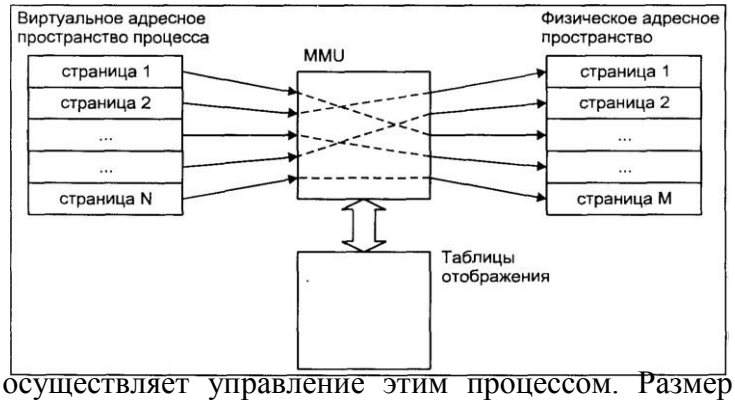
9. Принципы управления памятью. Виртуальная память. Страничная организация.

Принципы управления памятью

Подсистема управления памятью UNIX отвечает за справедливое и эффективное распределение разделяемого ресурса ОП м/у процессами и за обмен данными м/у ОП и вторичной памятью. Подсистема управления памятью современной многозадачной ОС должна обеспечивать:

- Выполнение задач, размер которых превышает размер ОП.
- Выполнение частично загруженных в память задач для минимизации времени их запуска.
- Размещение нескольких задач в памяти одновременно для повышения эффективности использования процессора.
- Размещение задачи в произвольном месте ОП.
- Размещение задачи в нескольких различных частях ОП.
- Совместное использование несколькими задачами одних и тех же областей памяти.

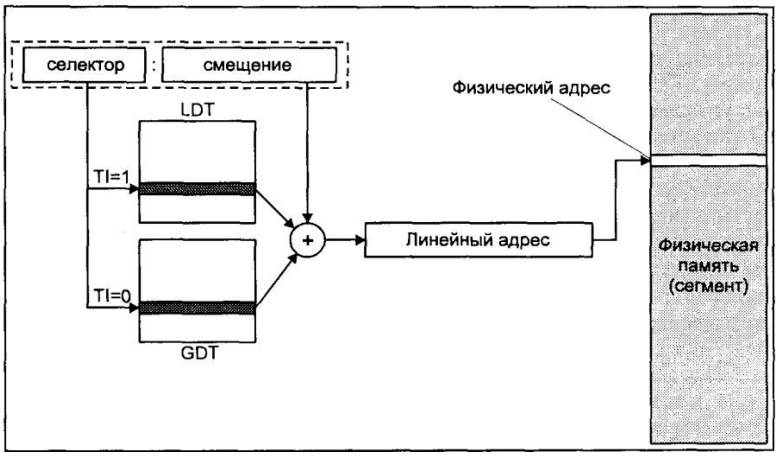
Все эти возможности реализованы в современных версиях UNIX с помощью *виртуальной памяти.* При использовании виртуальной памяти, адреса транслируются или отображаются в физические на аппаратном уровне при активном участии ядра ОС. Смысл виртуальной памяти – каждый процесс выполняется в собственном *виртуальном адресном пространстве, процессы* становятся изолированными друг от друга.



В современных компьютерных системах процесс отображения выполняется на аппаратном уровне (с помощью MMU) обеспечивая высокую скорость трансляции. ОС осуществляет управление этим процессом. Размер виртуальной памяти может существенно превышать размер физической за счет использования *вторичной памяти* или *области свопинга (*дискового пространства), где могут сохраняться временно не используемые участки адресного пространства процесса.

Современные процессоры поддерживают объединение адресного пространства в области переменного размера — *сегменты* и области фиксированного размера — *страницы.*

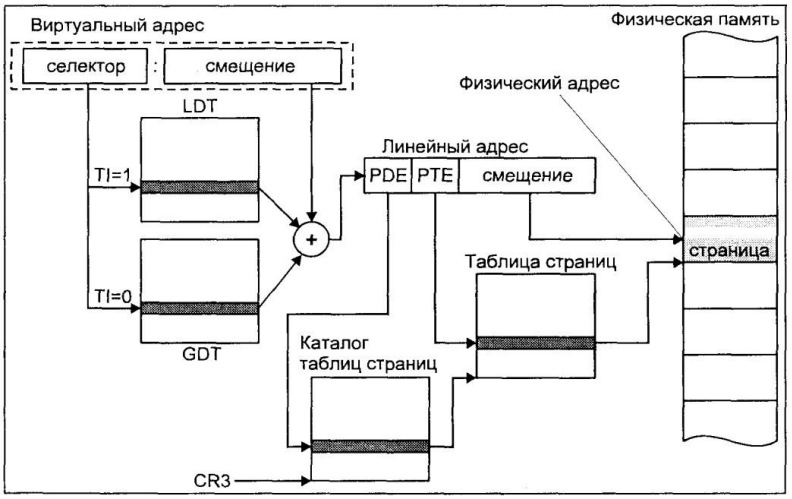
Семейство процессоров Intel позволяет разделить память на несколько логических частей – **сегменты.** При этом адресное пр-во процесса может быть представлено в виде нескольких логических сегментов, каждый из которых состоит из непрерывной последовательности адресов, лежащих в заданном диапазоне. Трансляция адресов, основанная на сегментации, предусматривает однозначное отображение адресов сегмента в непрерывную последовательность физических адресов. Виртуальный адрес при этом состоит из 2 частей: *селектора сегмента* и *смещения* относительно начала сегмента.



Селектор указывает на так называемый *дескриптор сегмента,* содержащий параметры, как его расположение в памяти, размер и права доступа. Дескрипторы сегментов расположены в 2 системных таблицах — *локальной таблице дескрипторов* (LDT) и *глобальной таблице дескрипторов* (GDT). LDT обеспечивает трансляцию вирт. адресов сегментов процесса; GDT обслуживает адресное пр-во ядра. Для каждого процесса создается собственная LDT, в то время как GDT разделяется всеми процессами. Инф-ия о таблице, на которую указывает селектор, комбинация селектора и смещения образует логический адрес.

Страничный механизм обеспечивает гораздо большую гибкость. Все виртуальное адресное пр-во разделено на блоки одинакового размера – *страницы***.** Основное преимущество такой схемы – система управления памятью оперирует областями достаточно малого размера для обеспечения эффективного распределения ресурсов памяти м/у процессами. Страничный механизм допускает, чтобы часть сегмента находилась в ОП, а часть отсутствовала. Это дает ядру возможность разместить в памяти только те страницы, которые в данное время используются процессом, тем самым значительно освобождая ОП.

Еще одним преимуществом – страницы сегмента могут располагаться в физической памяти в произвольном месте и порядке.



При использовании страничного механизма линейный адрес, полученный в результате сложения базового адреса сегмента и смещения также является логическим адресом, рассматривается процессором как состоящий из 3 частей. Первое поле адреса указывает на элемент *каталога таблиц страниц* (PDE). Каталог таблиц страниц имеет длину, равную одной странице, и содержит указатели на *таблицы страниц*. Второе поле адресует определенную страницу. Смещение на странице определяется третьим полем.

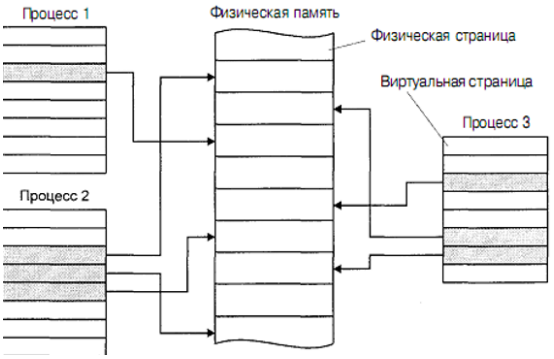
Блок страничной адресации процессора транслирует линейный адрес в физический. Использует поле PDE адреса в качестве индекса в каталоге таблиц. Элемент содержит адрес таблицы страниц. Второе поле линейного адреса PTE, позволяет процессору выбрать нужный элемент таблицы, адресующий физическую страницу. Складывая адрес начала страницы со смещением, хранящимся в третьем поле, процессор получает 32-битный физический адрес.

10. Основные алгоритмы замещения страниц. Понятие рабочего набора страниц.

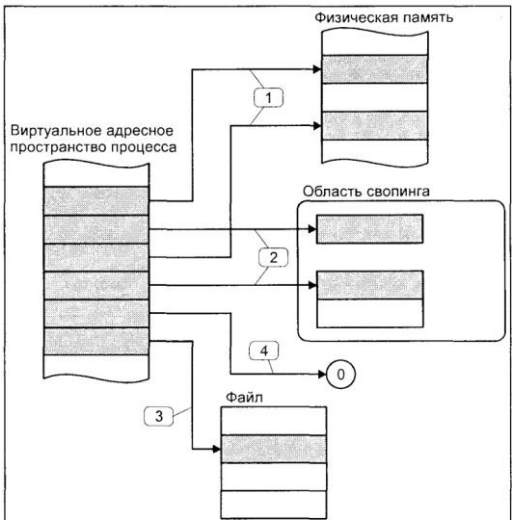
В системах с вирт. памятью, основанной на странич. механизме, адр. пр-во процесса разделено на последовательные участки равной длины – страницы. Любое место физ. памяти адресуется номером страницы и смещением в ней. Деление адр. пр-ва процесса явл-ся логическим, причем логическим последовательным страницам вирт. памяти при поддержке ОС и аппаратуры (MMU) ставятся в соответствие определенные физ. страницы ОП. Эта операция получила название *трансляции адреса.*

Но механизм трансляции адреса является *первым условием реализации вирт. памяти*, позволяя отделить вирт. адресное пр-во процесса от физ. адресного пр-ва процессора. *Вторым условием* яв-ся возможность выполнения процесса, чье адресное пр-во не имеет полного отображения на физ. память. Каждая страница вирт. памяти имеет флаг присутствия в ОП. Если адресуемая страница отсутствует в памяти, аппаратура генерирует страничную ошибку, которая обрабатывается ОС, в конечном итоге приводя к размещению этой страницы в памяти. Конкретный механизм страничного замещения зависит от того, как реализованы 3 основных принципа:

1. При каких условиях система загружает страницы в память, т.н. *принцип загрузки*.
2. В каких участках памяти система размещает страницы, т.н. *принцип размещения*.
3. Каким образом система выбирает страницы, которые требуется освободить из памяти, когда отсутствуют свободные страницы для размещения, т.н. *принцип замещения.*



В системах **с чистым страничным замещением по требованию** в память помещаются только требуемые страницы, а замещение производится, когда полностью отсутствует свободная ОП. Производительность таких систем полностью зависит от реализации принципа замещения. Однако большинство современных версий UNIX не используют данный принцип. Вместо него принцип загрузки предполагает **размещение сразу нескольких страниц, обращение к которым наиболее вероятно в ближайшее время**, а замещение производится до того, как память будет полностью занята.



Описанный механизм управления памятью допускает ситуацию, когда суммарный размер всех выполняющихся в данный момент процессов превышает размер физ. памяти, в которой располагается только часть страниц процессов. Содержимое остальных страниц хранится вне физ. памяти и должно быть загружено ядром, если процессу требуется доступ к этой части адр. пр-ва. Но вирт. адресное пр-во процесса не зависит от фактического расположения физических страниц, и его размещение производится ядром при создании процесса или запуске новой программы. Сам процесс "видит" только собственное вирт. адр. пр-во. Однако физические страницы, соответствующие этому адр. пр-ву могут в действительности располагаться в различных местах.

1. Виртуальный адрес может быть ассоциирован со страницей физической памяти.
2. Страница может быть перемещена в область свопинга, если требуется освободить память для другого процесса**.**
3. Адресуемая страница отсутствует в памяти, но ее содержимое находится в файле на диске.
4. Адресуемая страница отсутствует в памяти, и она не ассоциирована ни с областью свопинга, ни с файлом.

Наряду с картами необходимыми для трансляции адреса, ядро хранит ряд структур данных для поиска и загрузки отсутствующих в памяти страниц. Различные версии UNIX используют разные подходы.

Преимущества страничного замещения:

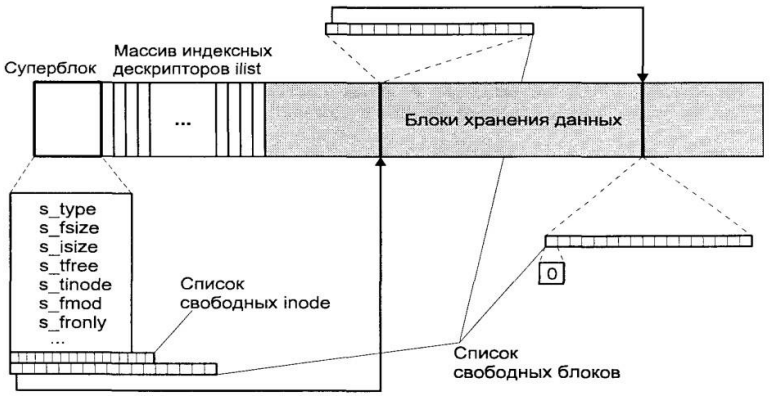
- Размер программы ограничивается лишь размером вирт. памяти, который для компьютеров с 32-разрядной архитектурой составляет 4 Гбайт.
- Запуск программы происходит очень быстро, т.к. не требуется загружать в память всю программу целиком.
- Значительно большее число программ может быть загружено и выполняться одновременно, т. к. для выполнения каждой из них в каждый момент времени достаточно всего нескольких страниц.
- Перемещение отдельных страниц между ОП и вторичной памятью требует значительно меньших затрат, чем перемещение процесса целиком.

Алгоритмы замещения страниц:

- Оптимальный алгоритм – выгрузить страницу, которая не будет использована больше всего;
- NRU (Not Recently Used) не использовавшаяся в последнее время страница;
- FIFO первым прибыл, первым обслужен – удаляется страница из головы списка;
- Вторая попытка (усовершествованный FIFO) – как и в фифо, но перед удалением проверяется спец. бит. Если он равен 0, то удаляется, иначе страница помещается в хвост;
- Часы (другая реализация алгоритма «Вторая попытка») – список страниц является кольцевым;
- LRU (least recently used) старница, не использовавшееся больше всего
- NFU (nor frequently used) редко использовавшаяся страница – удаляется страница с наименьшим значением счетчика;
- Старение – модернизация NFU со сдвигом счетчика вправо на 1 разряд перед прибавлением;
- Рабочий набор;
- WSClock.

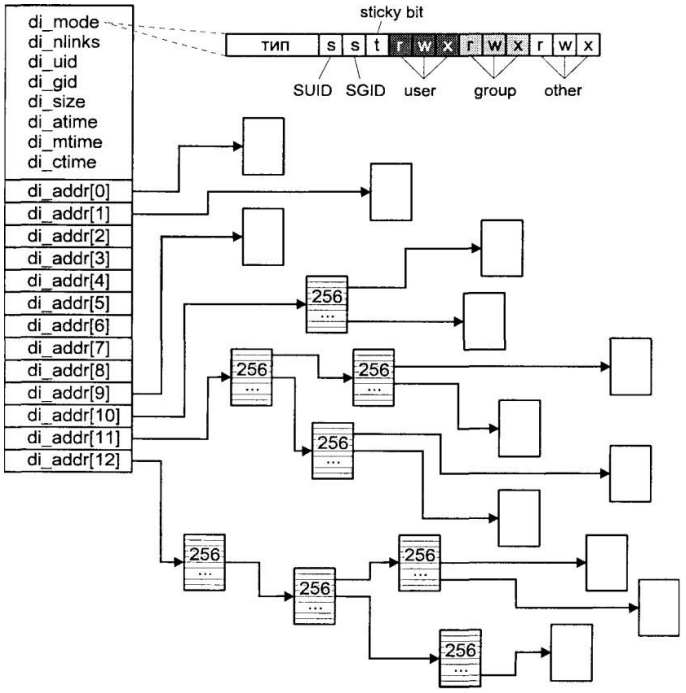
11. Структура файловой системы s5fs. Организация доступа к файлам.

Каждый жесткий диск состоит из одной или нескольких логических частей, называемых *разделами*. В разделе может располагаться только одна ФС, которая не может занимать несколько разделов. ФС s5fs занимает раздел диска и состоит из 3 основных компонентов.



Суперблок (СБ) содержит инф-ию, необходимую для монтирования и управления работой ФС в целом. В каждой ФС существует только один СБ, который располагается в начале раздела. СБ считывается в память при монтировании ФС и находится там до ее отключения. СБ содержит следующую информацию:

- Тип системы;
- Размер ФС в логических блоках, включая сам СБ, ilist и блоки хранения данных;
- Размер массива индексных дескрипторов;
- Число свободных блоков, доступных для размещения;
- Число свободных inode, доступных для размещения;
- Флаги;
- Размер логического блока (512, 1024, 2048);
- Список номеров свободных inode;
- Список адресов свободных блоков;



Индексные дескрипторы

Индексный дескриптор (inode) содержит *метаданные* файла. Каждый файл ассоциирован с одним inode, хотя может иметь несколько имен в ФС, каждое из которых указывает на один и тот же inode. При открытии файла ядро помещает копию дискового inode в память в таблицу incore inode, которая содержит несколько дополнительных полей. Основные поля дискового inode:

- Тип файла, дополнительные атрибуты выполнения и права доступа.
- Число ссылок на файл, т. е. количество которые имеет файл в файловой системе.
- Идентификаторы владельца - пользователя и владельца группы.
- Размер файла в байтах.
- Время последнего доступа к файлу.
- Время последней модификации.
- Время последней модификации inode (кроме модификации полей di_atime, di_mtime).
- Массив адресов дисковых блоков хранения данных.

Индексный дескриптор содержит инф-ию о расположении данных файла. Inode должен хранить физ. адреса всех блоков, принадлежащих данному файлу. В индексном дескрипторе эта инф-ия хранится в виде массива, каждый элемент которого содержит физ. адрес дискового блока, а индексом массива является номер логического блока файла. Массив имеет фиксированный размер и состоит из 13 элементов.

Файлы в UNIX могут содержать так называемые *дыры*. При чтении этой области процесс получит обнуленные байты. Поскольку логические блоки, соответствующие дыре, не содержат данные, не имеет смысла размещать для них дисковые блоки. В этом случае соответствующие элементы массива адресов inode содержат нулевой указатель. Когда процесс производит чтение такого блока, ядро возвращает последовательность нулей.

Имена файлов

Имя файла хранится в файлах специального типа — каталогах. Каталог файловой системы s5fs

представляет собой таблицу, каждый элемент которой имеет фиксированный размер в 16 байтов: 2 байта хранят номер индексного дескриптора файла, а 14 байтов — его имя. Это накладывает ограничение на число inode, которое не может превышать 65 535. Также ограничена и длина имени файла: его макс. размер — 14 символов. Размер каталога не уменьшается даже при удалении файлов.

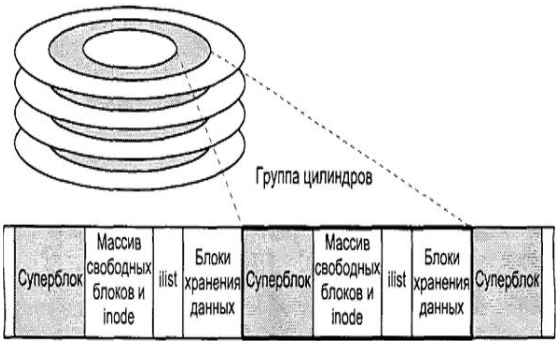
Недостатки и ограничения

С точки зрения надежности *слабым местом этой ФС является СБ*. СБ несет основную инф-ию о ФС в целом, и при его повреждении ФС не может использоваться.

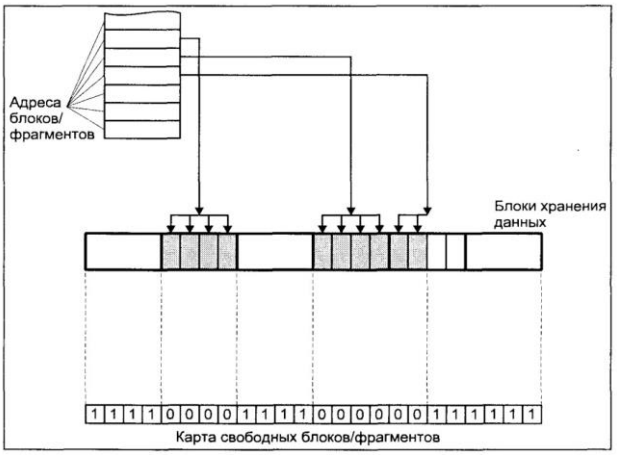
Относительно низкая производительность связана с размещением компонентов ФС на диске. Метаданные файлов располагаются в начале ФС, а далее следуют блоки хранения данных. При работе с файлом, происходит обращение как к его метаданным, так и к дисковым блокам, содержащим его данные. Поскольку эти структуры данных могут быть значительно разнесены в дисковом пространстве, необходимость постоянного перемещения головки диска увеличивает время доступа и, как следствие, уменьшает производительность ФС в целом.

Использование дискового пространства также не оптимально.

12. Структура файловой системы ffs. Организация каталога ffs.



Файловая система FFS обладает полной функциональностью системы s5fs, использует те же структуры данных ядра. Суперблок содержит общее описание ФС и располагается в начале раздела. Однако в суперблоке не хранятся данные о свободном пр-ве ФС, такие как массив свободных блоков и inode. Поэтому данные суперблока остаются неизменными на протяжении всего времени существования ФС. СБ дублируется для повышения надежности. Организация ФС предусматривает логическое деление дискового раздела на одну или несколько *групп цилиндров*. Группа цилиндров представляет собой несколько последовательных дисковых цилиндров. Каждая группа цилиндров содержит управляющую инф-ию, включающую резервную копию СБ, массив inode, данные о свободных блоках и итоговую информацию об использовании дисковых блоков в группе.



Для каждой группы цилиндров при создании ФС выделяется место под определенное количество inode. При этом обычно на каждые 2 Кбайт блоков хранения данных создается один inode. Идея такой структуры ФС заключается в создании кластеров inode, распределенных по всему разделу, вместо того, чтобы группировать все inode в начале. Управляющая инф-ия располагается с различным смещением от начала группы цилиндров. Это смещение выбирается равным одному сектору относительно предыдущей группы, таким образом для соседних групп управляющая инф-ия начинается на различных пластинах диска. Производительность ФС существенным образом зависит от размера блока хранения данных.
Чем больше размер блока, тем большее кол-во данных может быть прочитано без поиска и перемещения дисковой головки. ФС FFS поддерживает размер блока до 64 Кбайт. Каждый блок может быть разбит на 2, 4 или 8 фрагментов. В то время как блок является единицей передачи данных в операциях ввода/вывода, фрагмент определяет адресуемую единицу хранения данных на диске.
Инф-ия о свободном пространстве в группе хранится не в виде списка свободных блоков, а в виде *битовой карты блоков.*
Файловая система FFS при размещении блоков использует принципы:

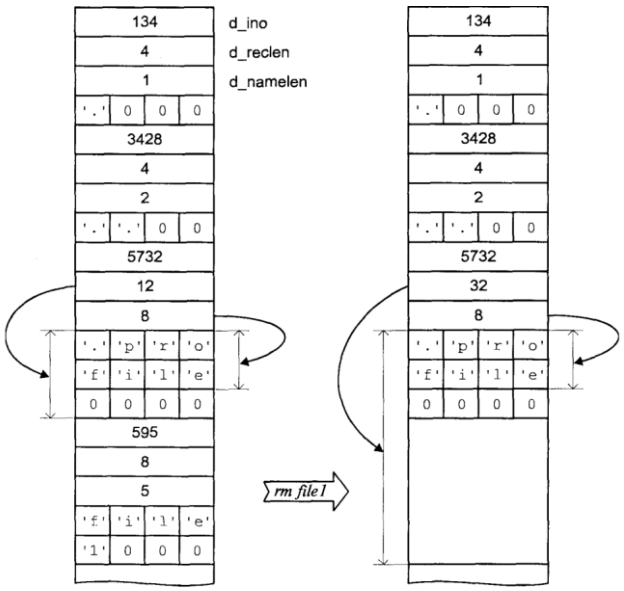
- Файл по возможности размещается в блоках хранения данных, принадлежащих одной группе цилиндров, где расположены его метаданные.
- Все файлы каталога по возможности размещаются в одной группе цилиндров.
- Каждый новый каталог по возможности помещается в группу цилиндров, от группы родительского каталога.
- Последовательные блоки размещаются исходя из оптимизации физического доступа.

Таким образом, правила размещения свободных блоков, с одной стороны, направлены на уменьшение времени перемещения головки диска, т. е. на локализацию данных в одной цилиндров, а с другой — на мерное распределение данных по диску.

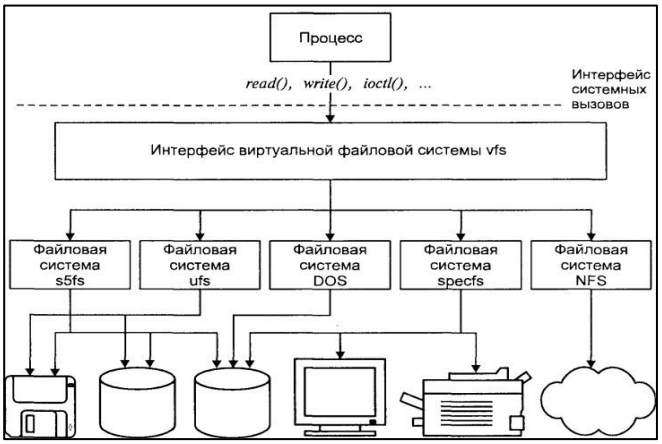
Каталоги. Структура каталога ФС FFS была изменена для поддержки длинных имен файлов (до 255 символов). Вместо записей фиксированной длины запись каталога FFS представлена структурой, имеющей следующие поля:

- Номер inode (индекс в массиве)
- Длина записи
- Длина имени файла
- Имя файла

Имя файла имеет переменную длину, дополненную нулями до границы. При удалении имени файла принадлежавшая ему запись присоединяется к предыдущей, и значение поля d reclen увеличивается на соответствующую величину. Удаление первой записи выражается в присвоении нулевого значения полю d_ino.

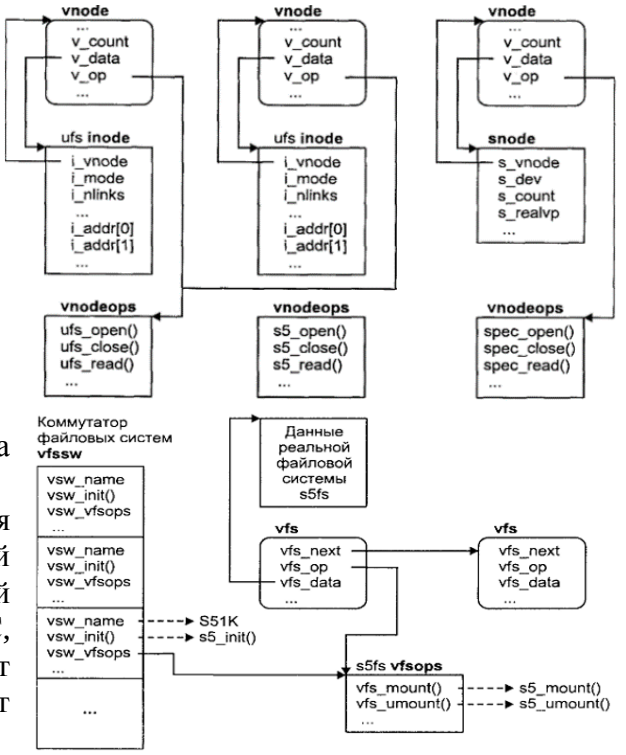

134
d_ino
4
d_reclen
1
d_namelen
'.' 0 0 0
3428
4
2
'.' '.' 0 0
5732
12
8
'.' 'p' 'r' 'o'
'f' 'i' 'l' 'e'
0 0 0 0
595
8
5
'f' 'i' 'l' 'e'
'1' 0 0 0
rm file1
134
4
1
'.' 0 0 0
3428
4
2
'.' '.' 0 0
5732
32
8
'.' 'p' 'r' 'o'
'f' 'i' 'l' 'e'
0 0 0 0

13. Виртуальная ФС. Монтирование файловой системы. Трансляция имени файла.



Современные версии UNIX обеспечивают одновременную работу с несколькими типами ФС путем разделения каждой ФС на *зависимый* и *независимый* от реализации уровни, последний из которых является общим и представляет для остальных подсистем ядра некоторую абстрактную ФС. Независимый уровень также называется *виртуальной ФС*.

Структура данных vnode одинакова для всех файлов, независимо от типа реальной ФС, где фактически располагается файл. Данные vnode содержат инф-ию, необходимую для работы виртуальной ФС, а также неизменные характеристики файла (тип файла). Взаимосвязь м/у независимыми дескрипторами (vnode) и зависимыми от реализации метаданными файла показана на рис.



Монтирование ФС. Прежде чем может состояться работа с файлами, соответствующая ФС должна быть встроена в существующее иерархическое дерево (операция *подключения* или *монтирования ФС)*. Каждая подключенная ФС представлена на независимом уровне в виде структуры vfs, аналоге записи таблицы монтирования дисковой ФС. Структуры vfs всех подключенных ФС организованы в виде односвязного списка, обеспечивая инф-ию, необходимую для обслуживания всего иерархического дерева, а также инф-ию о реальной ФС, которые не изменяются на протяжении работы.

Первой записью списка (таблицы монтирования) всегда явл-ся корневая ФС. Для инициализации и монтирования реальной ФС UNIX хранит *коммутатор файловых систем,* адресующий процедурный интерфейс для каждого типа ФС, поддерживаемой ядром. UNIX System V для этого использует глобальную таблицу, каждый элемент которой соответствует определенному типу реальной ФС, например s5fs, ufs или nfs.

Монтирование ФС производится системным вызовом mount(). В качестве аргументов передаются тип монтируемой ФС, имя каталога, к которому подключается ФС *(точка монтирования),* флаги и дополнительные данные, конкретный вид и содержимое которых зависят от реализации реальной ФС.

Трансляция имен. Ядро системы для обеспечения работы с файлами использует не имена, а индексные дескрипторы. Таким образом, необходима трансляция имени файла, передаваемого, например, в качестве аргумента системному вызову в номер соответствующего vnode. Системные вызовы, требующие трансляции имени: *exec()*; chown(); *chgrp()*; chmod(); statfs() – получение метаданных файла; rmdir( ) и т.д.

При этом полное имя может быть *абсолютным (*представляет корневой каталог, начинается с '/'*)* или *относительным.* Два каталога играют ключевую роль при трансляции имени: корневой каталог и текущий каталог. Каждый процесс адресует эти каталоги двумя полями структуры u_area:

- **Указатель текущего каталога**
- **Указатель на vnode корневого каталога**

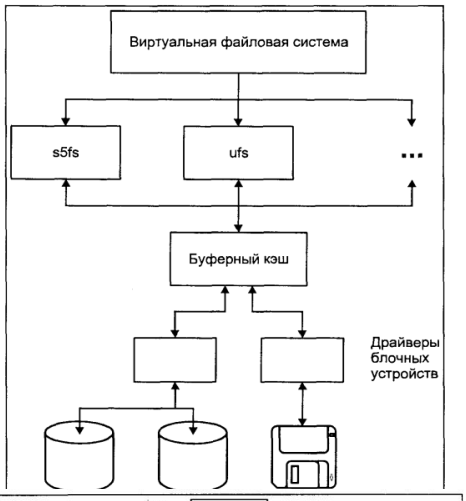
В зависимости от имени файла трансляция начинается с vnode, адресованного либо полем u_cdir либо u_rdir. Трансляция имени осуществляется покомпонентно, при этом для vnode текущего каталога вызывается соответствующая ему операция vn_lookup в качестве аргумента которой передается имя следующего компонента. В результате операции возвращается vnode, соответствующий искомому компоненту.

Если искомый файл является символической связью, и системный вызов, от имени которого

происходит трансляция имени, "следует" символической связи, операция vn_lookup() вызывает vn_readlink() для получения имени целевого файла.

Если оно является абсолютным (т. е. начинается с —/‖), то трансляция начинается с vnode корневого каталога, адресованного полем u_rdir области u_area. Процесс трансляции имени продолжается, пока не просмотрены все компоненты имени или не обнаружена ошибка.

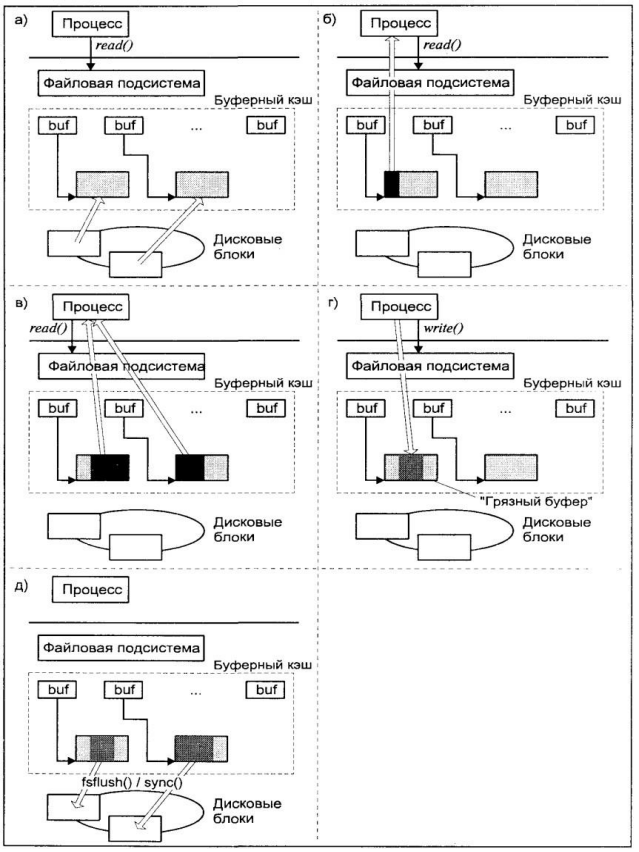
14. Буферный кэш. Реализация буферного кэша при помощи подсистемы управления памятью.



Работа файловой подсистемы тесно связана с обменом данными с периферийными устройствами. Для обычных файлов и каталогов — это устройство, на котором размещается соответствующая ФС, для специальных файлов устройств — это принтер, терминал, или сетевой адаптер.

Для повышения производительности дискового ввода/вывода и, соответственно, всей системы в целом, в UNIX используется кэширование дисковых блоков в памяти. Для этого используется выделенная область ОП, где кэшируются дисковые блоки файлов, к которым наиболее часто осуществляется доступ. Эта область памяти и связанный с ней процедурный интерфейс носят название *буферного КЭШа* и через него проходит большинство операций файлового ввода/вывода.

Буферный кэш состоит из буферов данных, размер которых достаточен для размещения одного дискового блока. С каждым блоком данных связан *заголовок буфера,* представленный структурой buf, с помощью которого ядро производит управление кэшем, включая идентификацию и поиск буферов, а также синхронизацию доступа. Заголовок также исп-ся при обмене данными с драйвером устройства для выполнения фактической операции ввода/вывода.



Буферный кэш использует механизм *отложенной записи*, при котором модификация буфера не вызывает немедленной записи на диск. Такие буферы отмечаются как "грязные", а синхронизация их содержимого с дисковыми данными происходит через определенные промежутки времени. Примерно 1/3 операций дискового ввода/вывода приходится на запись, причем один и тот же буфер может на протяжении ограниченного промежутка времени модифицироваться несколько раз. Поэтому буферный кэш позволяет значительно уменьшить интенсивность записи на диск и реорганизовать последовательность записи отдельных буферов для повышения производительности ввода/вывода. Однако этот механизм имеет свои недостатки, поскольку может привести к нарушению целостности ФС в случае неожиданного останова или сбоя ОС.

На рис. представлена схема выполнения операций ввода/вывода с использованием буферного кэша. Важной особенностью этой подсистемы – обеспечивает независимое выполнение операций чтения или записи данных процессом как результат соответствующих системных вызовов, а также фактический обмен данными с периферийным устройством. Когда процессу требуется прочитать или записать данные он использует системные вызовы *read()* или *write()*, направляя тем самым запрос файловой подсистеме. Файловая подсистема транслирует этот запрос в запрос на чтение или запись соответствующих дисковых блоков файла и направляет его в буферный кэш. Кэш просматривается на предмет наличия требуемого блока в памяти. Если соответствующий буфер найден, его содержимое копируется в адресное пр-во процесса в случае чтения и наоборот при записи, и операция завершается. Если блок в кэше не найден, ядро размещает буфер, связывает его с дисковым блоком с помощью заголовка buf и направляет запрос на чтение драйверу устройства.

Обычно используется схема чтения *вперед*, когда считываются не только запрашиваемые блоки, но и блоки, которые с высокой вероятностью могут потребоваться в ближайшее время (рис.а). последующие вызовы *read()* скорее всего не потребуют дискового ввода/вывода, а будут включать лишь копирование данных из буферов в память процесса.

При запросе на модификацию блока изменения также затрагивают только буфер кэша. При этом ядро помечает буфер как "грязный" в заголовке buf. Перед освобождением такого буфера для повторного использования, его содержимое должно быть предварительно сохранено на диске.

Перед фактическим использованием буфера доступ к нему для других процессов должен быть заблокирован. При обращении к уже заблокированному буферу процесс переходит в состояние сна, пока

данный ресурс не станет доступным. Не заблокированные буферы помечаются как свободные и помещаются в специальный список. Буферы в этом списке располагаются в порядке наименее частого использования. Таким образом, когда ядру необходим буфер, оно выбирает тот, к которому не было обращений в течение наиболее продолжительного промежутка времени. После того как работа с буфером завершена, он помещается в конец списка и является наименее вероятным кандидатом на освобождение и повторное использование. Поэтому, если процесс вскоре опять обратится к тому же блоку данных, операция вода/вывода по-прежнему будет происходить с буфером кэша. С течением времени буфер перемещается в направлении начала очереди, но при каждом последующем обращении к нему, будет помещен в ее конец.

Основной проблемой, связанной с буферным кэшем, является "старение" инф-ии, хранящейся в дисковых блоках, образы которых находятся в буферном кэше. При аварийном останове системы, это может привести к потере изменений данных файлов, сделанных процессами непосредственно перед остановом.

Для уменьшения вероятности таких потерь в UNIX имеется несколько возможностей:

1. Может использоваться системный вызов *sync(),* который обновляет все дисковые блоки, соответствующие "грязным" буферам. *sync(2)* не ожидает завершения операции ввода/вывода, таким образом после возврата из функции не гарантируется, что все "грязные" буферы сохранены на диске.
2. Процесс может открыть файл в синхронном режиме (указав флаг o_sync в системном вызове open()). При этом все изменения в файле будут немедленно сохраняться на диске. Наконец, через регулярные промежутки времени в системе пробуждается специальный системный процесс — диспетчер буферного кэша (fsflush или bdflush). Этот процесс освобождает "грязные" буферы, сохраняя их содержимое в соответствующих дисковых блоках (рис. д).

15. Нарушения целостности файловой системы. Методы их устранения

Значительная часть ФС находится в ОП. А именно, в ОП расположены суперблок примонтированной системы, метаданные активных файлов (в виде системно-зависимых inode и соответствующих им vnode) и даже отдельные блоки хранения данных файлов, временно находящиеся в буферном кэше.

Для ОС рассогласование м/у буферным кэшем и блоками хранения данных отдельных файлов не приведет к катастрофическим последствиям даже в случае внезапного останова системы, хотя с точки зрения пользователя все может выглядеть иначе. Содержимое отдельных файлов не вносит существенных нарушений в целостность ФС.

Другое дело, когда подобные несоответствия затрагивают метаданные файла или другую управляющую инф-ию ФС, например, суперблок. Многие файловые операции затрагивают сразу несколько объектов ФС, и, если на диске будут сохранены изменения только для части этих объектов, целостность ФС может быть существенно нарушена.

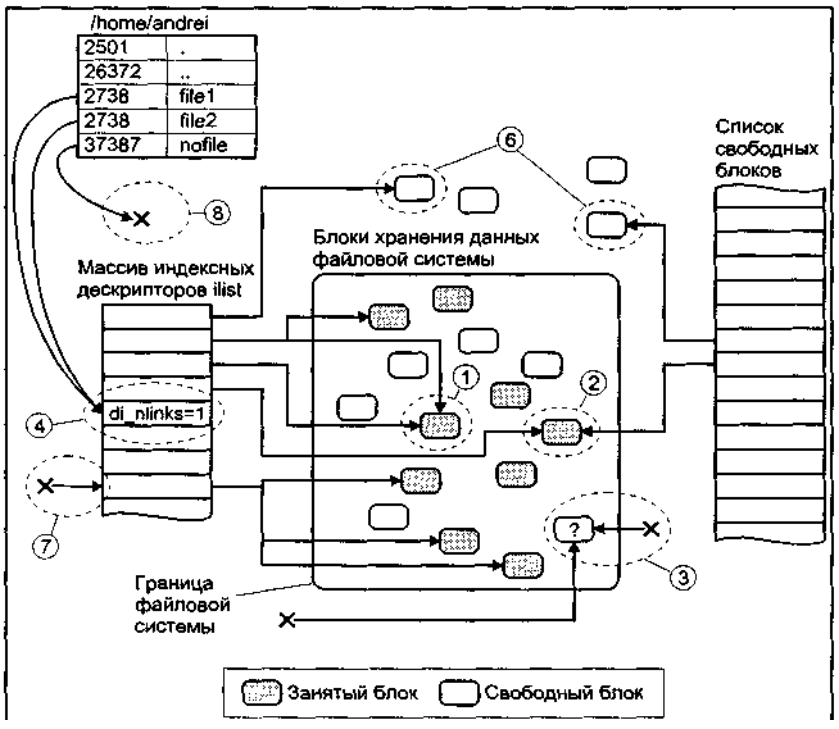
Рассмотрим пример создания жесткой связи для файла. Для этого файловой подсистеме необходимо выполнить следующие операции:

1. Создать новую запись в необходимом каталоге, указывающую на inode файла.
2. Увеличить счетчик связей в inode.

Предположим, что аварийный останов системы произошел м/у 1-ой и 2-ой операциями. В этом случае после запуска в ФС будут существовать 2 имени файла (2 записи каталогов), адресующие inode со счетчиком связей di_nlinks, равным 1. Если теперь будет удалено одно из имен, это приведет к удалению файла как такового, т.е. к освобождению блоков хранения данных и inode, поскольку счетчик связей di_nlinks станет равным 0. Оставшаяся запись каталога будет указывать на неразмещенный индексный дескриптор, или inode, адресующий уже другой файл.

Ядро выбирает порядок совершения операций с метаданными таким образом, чтобы вред от ошибок в случае аварии был минимальным. Однако проблема нарушения этого порядка все же остается, т. к. драйвер может изменять очередность выполнения запросов для оптимизации ввода/вывода. Единственной возможностью сохранить выбранный порядок является синхронизация операций со стороны файловой подсистемы.

В нашем примере файловая подсистема будет ожидать, пока на диск не будет записано содержимое индексного дескриптора, и только после этого произведет изменения каталога.



Отсутствие синхронизации м/у образом ФС в памяти и ее данными на диске в случае аварийного останова может привести к появлению следующих ошибок:

1) Один блок адресуется несколькими inode (принадлежит нескольким файлам).

2) Блок помечен как свободный, но в то же время занят (на него ссылается inode).

3) Блок помечен как занятый, но в то же время свободен (ни один inode на него не ссылается).

4) Неправильное число ссылок в inode (недостаток или избыток ссылающихся записей в каталогах).

5) Несовпадение между размером файла и суммарным размером адресуемых inode блоков.

6) Недопустимые адресуемые блоки (например, расположенные за пределами файловой системы).

7) "Потерянные" файлы (правильные inode, на которые не ссылаются записи каталогов).

8) Недопустимые или неразмещенные номера inode в записях каталогов.

Если нарушение все же произошло, на помощь может прийти утилита fsck(2) производящая исправление ФС. Запуск этой утилиты может производиться автоматически каждый раз при запуске системы, или администратором, с помощью команды*: fsck [options] filesystem,* где *filesystem* — специальный файл устройства, на котором находится ФС. Проверка и исправление должны производиться только на размонтированной ФС. Это связано с необходимостью исключения синхронизации таблиц в памяти (ошибочных) с их дисковыми эквивалентами (исправленными). Исключение составляет корневая ФС, кот. не может быть размонтирована. Для ее исправления необходимо исп-ть опцию – b обеспечивающую немедленный перезапуск системы после проведения проверки.

16. Сигналы. Диспозиция сигналов. Создание аварийного дампа памяти.

Сигнал является способом передачи уведомления о некотором произошедшем событии м/у процессами или м/у ядром системы и процессами. Сигналы можно рассматривать, как простейшую форму межпроцессного взаимодействия, хотя на самом деле они больше напоминают программные прерывания, при которых нарушается нормальное выполнение процесса.

Каждый сигнал имеет уникальное символьное имя и соответствующий ему номер. Напр-р, сигнал прерывания, посылаемый процессу при нажатии пользователем клавиши <*Del*> или <*Ctrl*>+<*C*>, имеет имя SIGINT. Сигнал, генерируемый комбинацией <*Ctrl*>+<\>, называется SIGQUIT.

Сигнал может быть отправлен процессу либо ядром, либо другим процессом с помощью системного вызова *kill(int pid, int sig).* Аргумент *pid* адресует процесс, которому посылается сигнал. Аргумент *sig* определяет тип отправляемого сигнала.

К генерации сигнала могут привести различные ситуации:

Ядро отправляет процессу (или группе процессов) сигнал при нажатии пользователем определенных клавиш или их комбинаций.

Аппаратные *особые ситуации (*деление на 0, обращение к недопустимой области памяти), также вызывают генерацию сигнала. Обычно эти ситуации определяются аппаратурой компьютера, и ядру посылается соответствующее уведомление (напр-р, в виде прерывания). Ядро реагирует на это отправкой соответствующего сигнала процессу, который находился в стадии выполнения, когда произошла особая ситуация.

Определенные программные состояния системы или ее компонентов также могут вызвать отправку сигнала. В отличие от предыдущего случая, эти условия не связаны с аппаратной частью, а имеют чисто программный характер.

С помощью системного вызова *kill()* процесс может послать сигнал как самому себе, так и другому процессу или группе процессов. В этом случае процесс, посылающий сигнал, должен иметь те же реальный и эффективный идентиф., что и процесс, которому сигнал отправляется. Разумеется, данное ограничение не распространяется на процессы, обладающие привилегиями супер-пользователя. Такие процессы имеют возможность отправлять сигналы любым процессам системы.

Процесс может выбрать одно из трех возможных действий при получении сигнала:

- игнорировать сигнал;
- перехватить и самостоятельно обработать сигнал;
- позволить действие по умолчанию.

Текущее действие при получении сигнала называется *диспозицией сигнала.* Напомним, что сигналы SIGKILL и SIGSTOP невозможно ни игнорировать, ни перехватить. Сигнал SIGKILL является силовым методом завершения выполнения "непослушного" процесса, а от работоспособности SIGSTOP зависит функционирование системы управления заданиями

При получении сигнала в большинстве случаев по умолч. происходит завершение выполнения процесса. В текущем рабочем каталоге процесса также создается файл **core,** в котором хранится образ памяти процесса. Этот файл может быть впоследствии проанализирован программой-отладчиком для определения состояния процесса непосредственно перед завершением. Файл **core** не будет создан в следующих случаях:

- исполняемый файл процесса имеет установленный бит SUID и реальный владелец-пользователь процесса не является владельцем-пользователем исполняемого файла;
- исполняемый файл процесса имеет установленный бит SGID, и реальный владелец-группа процесса не является владельцем-группой исполняемого файла;
- процесс не имеет права записи в текущем рабочем каталоге;
- размер файла **core** слишком велик.

Функция *signal(int sig, void (*disp(int)))* позволяет изменить диспозицию сигнала, которая по умолч. устанавливается ядром UNIX. Порожденный вызовом *fork()* процесс наследует диспозицию сигналов от своего родителя. Однако при вызове *exec(2)* диспозиция всех перехватываемых сигналов будет установлена на действие по умолчанию, т.к образ новой программы не содержит функции-обработчика, определенной диспозицией сигнала перед вызовом *exec().* Аргумент *sig* определяет сигнал, диспозицию которого нужно изменить. Аргумент *disp* определяет новую диспозицию сигнала, которой может быть определенная пользователем функция-обработчик или одно из следующих значений:

SIG_DFL Указывает ядру, что при получении процессом сигнала необходимо вызвать системный обработчик, т. е. выполнить действие по умолчанию.

SIG_IGN Указывает, что сигнал следует игнорировать.

17. Надежные и ненадежные сигналы. Блокирование доставки сигналов.

Ненадежные – Функция *signal(sig, handler)* определяется следующим образом: для указанного сигнала *sig* определяется поведение текущего процесса параметром *handler*, для которого возможны значения:

- SIG_IGN - игнорировать данный сигнал (кроме SIGKILL и SIGSTOP);
- SIG_DFL - использовать реакцию по умолчанию;
- указатель на функцию — обработчик сигнала (кроме SIGKILL и SIGSTOP).

Возвращаемым значением является указатель на предыдущий обработчик сигнала или SIG_ERR при ошибке. Для перехвата и собственной обработки сигнала необходимо определить функцию-обработчик сигнала типа void с единственным целым параметром, через который передается номер обрабатываемого сигнала — таким образом, можно использовать одну функцию-обработчик для нескольких различных сигналов.

Надежные. Функция *signal()* является ненадежной, поскольку не гарантирует выполнение обработчика при повторном получении сигнала и не контролирует прерывание системного вызова по сигналу, когда системный вызов возвращает код ошибки errno=EINTR. Это приводит к необходимости проверять значение глобальной переменной errno при каждом системном вызове.

Обработчик сигналов не должен делать никаких предположений о состоянии процесса при доставке ему сигнала, в частности, очень осторожно обращаться с глобальными переменными. Существует также список функций, которые небезопасно вызывать в обработчике сигналов.

Сигнал генерируется некоторым событием, но не может быть доставлен процессу мгновенно, а оказывается задержан в состоянии pending, которое может трансформироваться в состояние блокирован или доставлен. Блокированный сигнал не отбрасывается как игнорированный, но может быть доставлен и обработан позже, когда сигнал будет разблокирован процессом. Изменение маски осуществляется с помощью *sigprocmask()*. Такая задержка во времени не контролируется традиционной (устаревшей) обработкой сигналов.

Стандарт POSIX.1 определяет надежный способ обработки сигналов функцией *sigaction()* (и другими), где второй параметр либо замещается значением SIG_DFL или SIG_IGN, если сигнал не перехватывается, либо описывает действия по обработке сигнала. Третий параметр сохраняет предыдущее состояние условий обработки сигнала.

Блокирование. У программ также есть возможность приостановить обработку поступающих сигналов временно, на период выполнения какой-либо важной операции. В традиционной терминологии приостановка получения определенных сигналов называется блокированием. Если для поступившего сигнала было установлено блокирование, сигнал будет передан программе, как только она разблокирует данный тип сигналов. Этим блокирование отличается от игнорирования сигнала, при котором сигналы соответствующего типа никогда не передаются программе. Следует помнить, что не все сигналы могут быть проигнорированы.

18. Соглашения по группировке процессов. Организация процессов в группу лидера. Группировка в форме задания. Группировка в составе «сессии»

Группа процессов включает 1 или более процессов и существует, пока в системе присутствует хотя бы один процесс этой группы. Временной интервал, начинающийся с создания группы и заканчивающийся, когда последний процесс ее покинет, называется ***временем жизни группы.*** Последний процесс может либо завершить свое выполнение, либо перейти в другую группу.

Каждый процесс, помимо этого, является членом ***сеанса*** (**session**), являющегося набором 1 или нескольких групп процессов. Понятие сеанса было введено в UNIX для логического объединения процессов, а точнее, групп процессов, созданных в результате регистрации и последующей работы пользователя в системе. Таким образом, термин "сеанс работы" в системе тесно связан с понятием сеанса, описывающего набор процессов, которые порождены пользователем за время пребывания в системе.

Аргумент *pid,* который передается функции *getpgid(pid)* (получение id группы процесса, входящего в ту же сессию) адресует процесс, идентификатор группы которого требуется узнать. Если этот процесс не принадлежит к тому же сеансу, что и процесс, сделавший системный вызов, функция возвращает ошибку.
Системный вызов *setpgid(pid, pgid)* позволяет процессу стать членом существующей группы или создать новую группу. Функция устанавливает идентификатор группы процесса *pid* равным *pgid*. Процесс имеет возможность установить идентификатор группы для себя и для своих потомков (дочерних процессов). Однако процесс не может изменить идентификатор группы для дочернего процесса, который выполнил системный вызов *exec(),* запускающий на выполнение другую программу.
Если значения обоих аргументов равны, то создается новая группа с идентификатором *pgid*, а процесс становится *лидером* (group leader) этой группы. Поскольку именно таким образом создаются новые группы, их идентификаторы гарантированно уникальны. Заметим, что группа не удаляется при завершении ее лидера, пока в нее входит хотя бы один процесс.

Идентификатор сеанса можно узнать с помощью функции *getsid(pid).* Как и в случае с группой, идентификатор *pid* должен адресовать процесс, являющийся членом того же сеанса, что и процесс, вызвавший *getsid().*
Данные ограничения не распространяются на процессы, имеющие привилегии суперпользователя.

Вызов функции *setsid()* приводит к созданию **нового сеанса.**
Понятия группы и сеанса тесно связаны с терминалом или, точнее, с драйвером терминала. Каждый сеанс может иметь один ассоциированный терминал, который называется *управляющим терминалом,* а группы, созданные в данном сеансе, наследуют этот управляющий терминал. Наличие управляющего терминала позволяет ядру контролировать стандартный ввод/вывод процессов, а также дает возможность отправить сигнал всем процессам ассоциированной с терминалом группы, например, при его отключении.

Типичным примером является регистрация и работа пользователя в системе. При входе в систему терминал пользователя становится управляющим для лидера сеанса (в данном случае для командного интерпретатора shell) и всех процессов, порожденных лидером (в данном случае для всех процессов, которые запускает пользователь из командной строки интерпретатора). При выходе пользователя из системы shell завершает свою работу и таким образом отключается от управляющего терминала, что вызывает отправление сигнала SIGHUP всем незавершенным процессам текущей группы. Это гарантирует, что после завершения работы пользователя в системе не останется запущенных им процессов.

19. Неименованные каналы. Организация взаимодействия процессов при помощи неименованных каналов. Конвейер команд.

Неименованный канал является средством взаимодействия м/у связанными процессами – родительским и дочерним. Родительский процесс создает канал при помощи системного вызова: **int pipe(int fd[2])**, который возвращает два файловых дескриптора — fildes[0] для записи в канал и fildes[1] для чтения из канала. Теперь, если один процесс записывает данные в fildes[0], другой сможет получить эти данные из fildes[1]. Дочерний процесс наследует и разделяет все назначенные файловые дескрипторы родительского. То есть доступ к дескрипторам fildes канала может получить сам процесс, вызвавший **pipe**, и его дочерние процессы. В этом заключается серьезный недостаток каналов, т.к. они могут быть использованы для передачи данных только между родственными процессами. Каналы не могут использоваться в качестве средства межпроцессного взаимодействия между независимыми процессами.

Если нужен двунаправленный обмен данными м/у процессами, то родительский процесс создает 2 канала, один из которых используется для передачи данных в одну сторону, а другой - в другую. После получения процессами дескрипторов канала для работы с каналом используются файловые системные вызовы:
int **read**(int pipe_fd, void *area, int cnt); int **write**(int pipe_fd, void *area, int cnt);
Первый аргумент этих вызовов - дескриптор канала, второй - указатель на область памяти, с которой происходит обмен, третий - количество байт. Оба вызова возвращают число переданных байт (или -1 - при ошибке).

Каналы работают по следующим правилам:

1. При чтении меньшего числа байтов, чем находится в канале, возвращается требуемое число байтов, остаток сохраняется для последующих чтений.
2. При чтении большего числа байтов, чем находится в канале, возвращается доступное число байтов.
3. Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено 0 байтов. Если один или более процессов открыли канал для записи, вызов будет заблокирован до появления данных (если для канала не установлен флаг отсутствия блокирования O_NDELAY).
4. Запись числа байтов, меньшего емкости канала, гарантированно атомарно (в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются).
5. При записи большего числа байтов, чем это позволяет, вызов блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним процессом на чтение, процессу генерируется сигнал SIGPIPE, а вызов возвращает 0 с установкой ошибки (если процесс не установил обработки SIGPIPE, то обработка по умолчанию – завершение).

Передача потока вывода одной программы в поток ввода другой осуществляется с помощью **конвейера (программного канала)**. Он обеспечивает однонаправленную передачу данных м/у двумя задачами. Программные каналы часто используются для фильтрации вывода некоторой команды: **$ cat myfile | wc**. Вывод программы cat, которая выводит содержимое файла **myfile,** передается на ввод программы wc, которая подсчитывает количество строк, слов и символов. В результате мы получим что-то вроде: 12 45 260, что будет означать количество строк, слов и символов в файле **myfile.**

20. Именованные каналы, правила и особенности их использования. Синхронизация процессов, использующих именованные каналы.

Именованные каналы (FIFO) могут использоваться как средство взаимодействия м/у неродственными и удаленными процессами. Такой канал имеет внешнее имя, которое включается в пространство имен ФС. FIFO является отдельным типом файла в ФС UNIX (ls –l покажет символ *p* в первой позиции). Для создания FIFO используется системный вызов *int mknod(char *pathname, int mode, int dev);* где *pathname* – указатель на строку, содержащую имя файла в ФС (имя FIFO), *mode* — флаги владения, прав доступа и т. д., *dev* — при создании FIFO игнорируется.
После создания FIFO может быть открыт на запись и чтение, причем запись и чтение могут происходить в разных независимых процессах.

При работе с именованным каналом используются файловые системные вызовы: *int open(int *name, int oflag); int read(int pipe_fd, void *area, int cnt); int write(int pipe_fd, void *area, int cnt); int close(int pipe_fd);*

Именованный канал является постоянным объектом, он сохраняется после завершения создавшего его процесса и при необходимости должен быть уничтожен явно - при помощи системного вызова: *int unlink(char *name).*

Через канал может быть передан неограниченный объем инф-ии, хотя мгновенная емкость канала ограничена 10-ю блоками. Ситуации переполнения канала при записи и чтение пустого канала автоматически блокируются скрытым **механизмом синхронизации обмена.** Он обеспечивает приостановку процесса записи, когда в канале нет места для размещения новых данных, или процесса чтения, при попытке ввода из пустого канала, который пока открыт по записи.

Каналы FIFO работают по следующим правилам:

1. При чтении меньшего числа байтов, чем находится в канале FIFO, возвращается требуемое число байтов, остаток сохраняется для последующих чтений.
2. При чтении большего числа байтов, чем находится в канале FIFO, возвращается доступное число байтов.
3. Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено 0 байтов. Если один или более процессов открыли канал для записи, вызов будет заблокирован до появления данных (если для канала FIFO не установлен флаг отсутствия блокирования O_NDELAY).
4. Запись числа байтов, меньшего емкости канала FIFO, гарантированно атомарно (в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются).
5. При записи большего числа байтов, чем это позволяет FIFO, вызов блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним процессом на чтение, процессу генерируется сигнал SIGPIPE, а вызов возвращает 0 с установкой ошибки (если процесс не установил обработки SIGPIPE, то обработка по умолчанию – завершение).

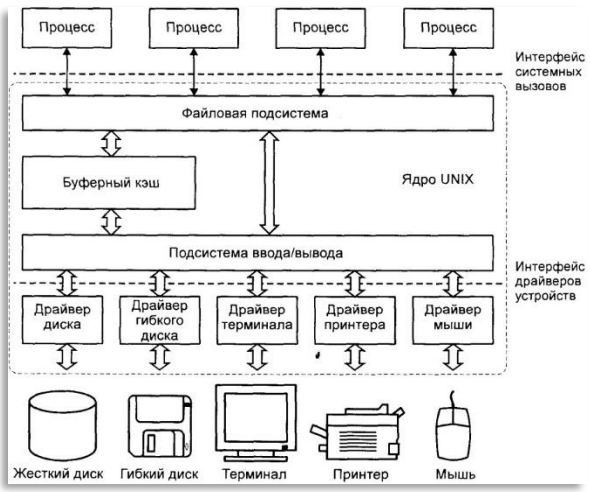
21-22. Подсистема ввода/вывода. Коммутатор устройств. Драйверы устройств

Подсистема ввода/вывода выполняет запросы файловой подсистемы и подсистемы управления процессами для доступа к периферийным устройствам (дискам, магнитным лентам, терминалам и т. д.). Она обеспечивает необходимую буферизацию данных и взаимодействует с **драйверами устройств** *(это основные компоненты подсистемы)* — специальными модулями ядра, обеспечивающими непосредственную работу с периферийными устройствами.

Драйверы устройств обеспечивают интерфейс м/у ядром UNIX и аппаратной частью компьютера. Благодаря этому от остальной части ядра скрыты архитектурные особенности компьютера, что значительно упрощает перенос системы и поддержку работы различных периферийных устройств.

В UNIX существует большое кол-во драйверов. Часть из них обеспечивает доступ к физическим устройствам (жесткому диску, принтеру или терминалу), другие предоставляют аппаратно-независимые услуги (драйверы для работы с виртуальной памятью ядра и представляющий "нулевое" устройство).
В процессе запуска системы ядро вызывает соответствующие процедуры инициализации установленных драйверов.

Типы драйверов: драйверы различаются по возможностям, которые они предоставляют, по тому, каким образом обеспечивается к ним доступ и управление.
Символьные драйверы обеспечивают работу с устройствами с побайтовым доступом и обменом данными (символьно-ориентированные устройства). Это модемы, терминалы, принтеры, мышь и т. д. Доступ к таким драйверам не включает использование буферного кэша, таким образом, ввод и вывод не буферизуется.
Блочные драйверы позволяют производить обмен данными с устройством (блочно- ориентированным) фиксированными порциями (блоками). Напр-р, для жесткого диска данные можно адресовать и читать только секторами, размер которых составляет несколько сотен байтов. Для блочных драйверов обычно используется буферный кэш, который и является интерфейсом м/у ФС и устройством. Хотя операции чтения и записи для процесса допускают обмен данными, размер которых меньше размера блока, на системном уровне это все равно приводит к считыванию всего блока, изменению части его данных и записи измененного блока обратно на диск.
Драйверы низкого уровня (raw drivers). Этот тип интерфейса блочных драйверов позволяет производить обмен данными с блочными устройствами, минуя буферный кэш. Это означает, что устройство может быть адресовано элементами, размер которых не совпадает с размером блока. Обмен данными происходит независимо от файловой подсистемы и буферного кэша, что позволяет ядру производить передачу м/у пользовательским процессом и устр-вом без дополнительного копирования.
На рис. **схема взаимодействия драйверов устройств** с другими подсистемами UNIX.



Драйвер устройства адресуется **старшим номером** (major number) устройства. Среди атрибутов специальных файлов устройств, которые обеспечивают пользовательский интерфейс доступа к периферии компьютера также есть еще и **младший номер (minor number**). Младший номер интерпретируется самим драйвером (например, для клонов, оно задает старшее число устройства, которое требуется "размножить").

Доступ к драйверу осуществляется ядром через специальную структуру данных – **коммутатор устройств**, каждый элемент которой содержит указатели на соответствующие функции драйвера – точки входа. Старшее число, по существу, является указателем на элемент коммутатора устройств, обеспечивая тем самым ядру возможность вызова необходимой функции указанного драйвера.

Таким образом, коммутатор устройств определяет базовый интерфейс драйвера устройств. Ядро содержит коммутаторы устройств двух типов: **bdevsv** для блочных и **cdevsw** для символьных устройств. Ядро размещает отдельный массив для каждого типа коммутатора, и любой драйвер устройства имеет запись в соответствующем массиве. Если драйвер обеспечивает как блочный, так и символьный интерфейсы, его точки входа будут представлены в обоих массивах.

БОЛЬШЕ ОТНОСИТСЯ К 22 БИЛЕТУ:
Не все драйверы служат для работы с физическими устройствами, такими как сетевой адаптер, последовательный порт или монитор. Часть драйверов служат для предоставления различных услуг ядра прикладным процессам и не имеют непосредственного отношения к аппаратной части компьютера. Такие

драйверы называются **программными или драйверами псевдоустройств**. Можно привести несколько примеров псевдоустройств и соответствующих им программных драйверов:
/dev/kmem: Обеспечивает доступ к виртуальной памяти ядра. Зная виртуальные адреса внутренних структур ядра, процесс может считывать хранящуюся в них информацию. С помощью этого драйвера может, например, быть реализована версия утилиты ps(1), выводящей информацию о состоянии процессов в системе.
/dev/ksyms: Обеспечивает доступ к разделу исполняемого файла ядра, содержащего таблицу символов. Совместно с драйвером **/dev/kmem** обеспечивает удобный интерфейс для анализа внутренних структур ядра.
/dev/mem: Обеспечивает доступ к физической памяти компьютера.
/dev/null: Является "нулевым" устройством. При записи в это устройство данные просто удаляются, а при чтении процессу возвращается 0 байтов. Пр-р использования: подавление вывода сообщений об ошибках.
/dev/zero: Обеспечивает заполнение нулями указанного буфера. Этот драйвер часто используется для инициализации области памяти.

23. Архитектура терминального доступа. Псевдотерминал. Прозрачный и канонический режимы работы драйвера терминала.

Алфавитно-цифровой терминал – последовательное устройство; ОС производит обмен данными с терминалом через последовательный интерфейс, называемый **терминальной линией**. С каждой терминальной линией в UNIX ассоциирован специальный файл символьного устройства **/dev/ttyxx (**в зависимости от версии UNIX вместо символов *xx в* имени файла терминала присутствует идентификатор, позволяющий поставить в соответствии файлу конкретную терминальную линию**).** Терминальные драйверы выполняют ту же ф-ию, что и остальные драйверы: **управление передачей данных от/на терминалы.** Но терминалы имеют одну особенность, связанную с тем, что они обеспечивают интерфейс пользователя с системой. Обеспечивая интерактивное использование системы UNIX, терминальные драйверы имеют свой внутренний интерфейс с модулями, интерпретирующими ввод и вывод строк. Модуль, отвечающий за такую обработку, называется **дисциплиной линии.**

Существует два режима терминального ввода/вывода:
1. **Канонический режим**. Ввод с терминала обрабатывается в виде законченных строк.
2. **Неканонический режим.** Ввод не интерпретируется.

В каноническом режиме интерпретаторы строк преобразуют неструктурированные последовательности данных, введенные с клавиатуры, в каноническую форму (в форму, соответствующую тому, что пользователь имел в виду на самом деле) прежде, чем послать эти данные принимающему процессу. Напр-р, программисты работают на клавиатуре терминала быстро, но допускают ошибки. На этот случай есть клавиша стирания. Драйвер терминала получает всю введенную последовательность, включая символы стирания. В каноническом режиме модуль дисциплины линии буферизует инф-ию в строку (набор символов, заканчивающийся символом возврата каретки) и стирает символы в буфере, прежде чем переслать исправленную последовательность считывающему процессу. В таком режиме работает командный интерпретатор shell.

В режиме без обработки строковый интерфейс передает данные между процессами и терминалом без каких-либо преобразований.

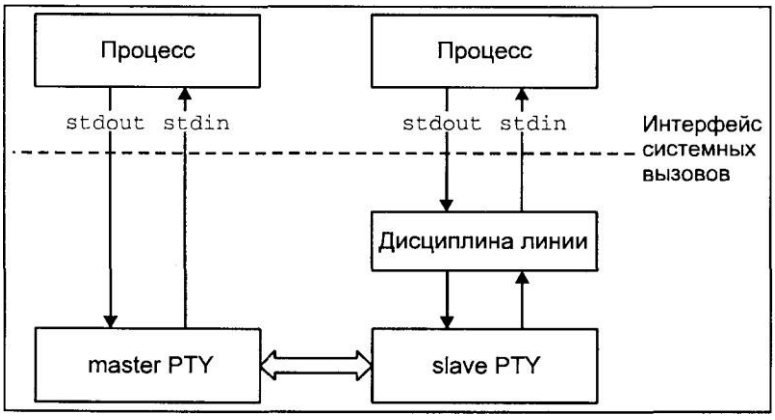
В **функции модуля дисциплины** линии входят:
1. Построчный разбор введенных последовательностей.
2. Обработка символов стирания.
3. Обработка символов удаления, отменяющих все предыдущие символы.
4. Отображение символов, полученных терминалом.
5. Расширение выходных данных (преобразование символов табуляции в последовательности пробелов).
6. Предоставление возможности не обрабатывать специальные символы, такие как символы стирания, удаления и возврата каретки.

Есть возможность обработки данных, получаемых и передаваемых устройству – отображение символов в символы, определенные **таблицей отображения** (утилита **mapchan**).

Псевдотерминалы являются специальным устройством, эмулирующим стандартную терминальную линию. Псевдотерминалы напоминают каналы как средство межпроцессного взаимодействия, позволяющее двум процессам обмениваться данными. Однако в отличие от каналов, псевдотерминалы обеспечивают дополнительную функциональность, специфичную для терминальных линий.

Примером использования псевдотерминалов является регистрация в системе по сети с использованием серверов удаленного доступа **rlogin** или **telnet**. Когда пользователь регистрируется в системе подобным образом, псевдотерминал эмулирует обычную терминальную линию, поэтому пользователь не видит различия м/у удаленной и локальной работой с помощью терминала, подключенного по последовательной линии.

Схематически арх-ра псевдотерминала представлена на рис.



Псевдотерминал по существу представляет собой два отдельных драйвера. Один из них выглядит как обычный терминальный драйвер и носит название **подчиненного устройства (slave).** Второй драйвер называется **основным.** Поскольку подчиненное устройство имеет все характеристики терминала, процесс

может связать свои стандартные потоки ввода, вывода и вывода ошибок с этим устройством. Однако в отличие от обычного терминала, в случае которого запись процесса приводит к отображению данных на физическом устройстве, а ввод данных пользователем с клавиатуры может быть получен чтением терминальной линии, все данные, записанные в подчиненное устройство, передаются основному и наоборот — почти так, как работает канал. Однако модуль дисциплины линии позволяет обеспечить дополнительные возможности этого канала, которые могут потребоваться некоторым приложениям, например, командному интерпретатору shell.

24. Общие принципы организации и использования средств IPC SystemV

Множество возможных имен объектов конкретного типа межпроцессного взаимодействия называется **пространством имен**. Имена являются важным компонентом системы межпроцессного взаимодействия для всех объектов, т.к. позволяют различным процессам получить доступ к общему объекту. Для таких объектов IPC, как **очереди сообщений, семафоры и разделяемая память**, процесс назначения имени является более сложным, чем просто указание имени файла. Имя для этих объектов называется **ключом** (key) и генерируется функцией **ftok** из двух компонентов — имени файла и отличного от нуля id проекта:
key_t ftok(const char *pathname, int id);
В качестве pathname можно использовать маршрутное имя некоторого файла, известное взаимодействующим процессам (имя программы-сервера). Важно, чтобы этот файл существовал на момент создания ключа.

Пр-во имен позволяет создавать и совместно использовать IPC неродственным процессам. Но для ссылок на созданные объекты используются id. IPC имеет свой уникальный дескриптор (идентификатор), используемый ОС (ядром) для работы с объектом. Уникальность дескриптора обеспечивается уникальностью дескриптора для каждого типа объектов (очереди сообщений, семафоры и разделяемая память), т. е. какая-либо очередь сообщений может иметь тот же численный идентификатор, что и разделяемая область памяти (но 2 очереди сообщений должны иметь различные идентификаторы).

Для создания или получения доступа к объекту используются системные вызовы **get: msgget** для очереди сообщений, **semget** для семафора и **shmget** для разделяемой памяти. Все эти вызовы возвращают дескриптор объекта в случае успеха и -1 в случае неудачи. Ф-ии **get** позволяют процессу получить ссылку на объект (возвращаемый дескриптор), но не позволяют производить операции над ним. **get** в качестве аргументов используют **key** и флаги создания объекта **ipcflag**. Остальные аргументы зависят от типа объекта. Переменная ipcflag - права доступа к объекту PERM (например 0400 как r-------- - чтение для владельца), и указывает, создается новый объект или требуется доступ к существующему (IPC_CREAT и IPC_EXCL).

Работа с объектами IPC System V похожа на работу с файлами в UNIX. Одно из различий - файловые дескрипторы имеют значимость в контексте процесса, а значимость дескрипторов объектов IPC распространяется на всю систему. Для каждого из объектов IPC ядро поддерживает структуру данных, отличную для каждого типа объекта (очереди сообщений, семафора или разделяемой памяти). Общей у этих данных является **структура, описывающая права доступа к объекту** (как для файлов):
uid - идентификатор владельца-пользователя объекта; **gid** - id владельца-группы объекта
suid - UID создателя объекта; **cgid** - GID создателя объекта
mode - права доступа на чтение и запись для всех классов доступа (9 битов)
key - ключ объекта

Права доступа (как и для файлов) определяют возможные операции, выполняемые над объектом конкретным процессом (получение доступа к существующему объекту, чтение, запись и удаление). Система не удаляет созданные объекты IPC даже тогда, когда ни один процесс не пользуется ими. С помощью функций управления **msgctl**, **semctl, shmctl** процесс может получить и установить ряд полей внутренних структур, поддерживаемых системой для объектов, а также удалить созданные объекты.

25. Сокеты. Коммуникационный домен. Привязка сокета

Сокет — коммуникационный узел, обеспечивающий прием и передачу данных для объекта (процесса).
Коммуникационный домен — понятие, описывающее набор таких характеристик межпроцессного взаимодействия как схема адресации объектов, их расположение, протоколы передачи данных и т.д.

Сокеты создаются в рамках определенного коммуникационного домена, подобно тому, как файлы создаются в рамках ФС. Сокеты имеют соответствующий интерфейс доступа к ФС UNIX и так же, как обычные файлы, адресуются некоторым целым числом — дескриптором. Но, в отличие от обычного файла, сокет представляет собой виртуальный объект, который существует, пока на него ссылается хотя бы 1 процесс.

В зависимости от предоставляемых коммуникационных характеристик сокеты делятся на след. **типы**:
1) сокет дейтаграмм (дейтаграммный сокет, datagram socket, SOCK_DGRAM). Теоретически ненадежная, несвязная передача пакетов.
2) сокет потока (потоковый сокет, stream socket, SOCK_STREAM). Надежная передача потока байтов без сохранения границ сообщений. Поддерживает передачу экстренных данных.
3) сокет пакетов (пакетный сокет, packet socket, SOCK_SEQPACKET). Надежная передача данных без дублирования с предварительным установлением связи. При этом сохраняются границы сообщений
4) сокет низкого уровня (raw socket, SOCK_RAW). Через него осуществляется непосредственный доступ к коммуникационному протоколу.

Экстренные сообщения доставляются вне нормального потока данных. Они связаны с некими срочными событиями, требующими немедленной реакции.
Для сокетов определено пространство имен, чтобы независимые процессы могли взаимодействовать друг с другом. Имя сокета имеет смысл только в рамках коммуникационного домена, в котором он создан. Имена сокетов представлены адресами. Таким образом, сокет – это коммуникационный интерфейс взаимодействующих процессов. Конкретный характер взаимодействия зависит от используемых сокетов, а коммуникационный домен, в рамках которого создан сокет, определяет базовые св-ва этого взаимодействия.

Для создания сокета процесс должен указать тип сокета и коммуникационный домен, в рамках которого будет использоваться сокет. Поскольку коммуникационный домен может поддерживать использование нескольких протоколов, процесс может (но не обязан) также указать конкретный коммуникационный протокол для взаимодействия.
Системный вызов, создающий сокет: ***int socket (int domain, int type, int protocol);***
Коммуникационный домен определяет семейство протоколов, допустимых в рамках данного домена. Возможные значения аргумента domain включают:
1) AF_UNIX. Домен локального межпроцессного взаимодействия в пределах единой ОС UNIX. Внутренние протоколы.
2) AF_INET. Домен взаимодействия процессов удаленных систем. Протоколы Internet (TCP/IP).
3) AF_NS. Домен взаимодействия процессов удаленных систем. Протоколы Xerox NS.

Домен может не поддерживать определенные типы сокетов: так, AF_UNIX не поддерживает пакетные и низкоуровневые сокеты, а AF_INET — пакетные.
Фактической передаче данных предшествует фаза **связывания** (binding) сокета, когда устанавливается дополнительная инф-ия, необходимая для определения коммуникационного узла. ***int bind (int sockfd, struct sockaddr * localaddr, int addrlen);***
sockfd – дескриптором сокета, полученным при его создании; *localaddr* – локальный адрес, с которым необходимо связать сокет; *addrlen* – размер адреса.

В отличие от локального межпроцессного взаимодействия, сетевой обмен данными нуждается в указании как локального процесса, так и хоста, на котором выполняется данный процесс.