Министерство образования Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

В.М. Стасышин, Д.С. Ильиных,
К.И. Воронов

Управление ресурсами в вычислительных системах

Конспект лекций
для студентов III курса
специальности 510200
(Электронная версия)

Новосибирск
2004

Статышин В.М., Ильиных Д.С., Воронов К.И. Управление ресурсами в ВС: Конспект лекций. – Новосибирск, 2004 – 106 с .

Утверждено Редакционно-издательским советом группы ПМ-13 в качестве конспекта лекций для студентов третьего курса факультета прикладной математики и информатики.

Курс лекций по управлению ресурсами опирается на курсы лекций по программному обеспечению вычислительных систем и основам сетевых технологий, поскольку многие идеи, изложенные в настоящем пособии, основаны на знаниях, полученных по вышеперечисленным базовым дисциплинам. Цель данной работы – на доступном уровне изложить основные теоретические понятия, на которых строится работа большинства известных операционных систем, помочь в решении практических задач, неизбежно возникающих при более детальном знакомстве с операционными системами «изнутри». К сожалению, в данное пособие не включены теоретические вопросы, затрагивающие вопрос управления ресурсами в ОС MS-DOS, поскольку данный конспект был подготовлен за очень короткое время. Вопросы по ОС MS-DOS (которые, возможно, будут освещены в дальнейших редакциях), предлагаемые на самостоятельное изучение, вынесены в приложение к данному конспекту лекций.

Курс лекций предназначен для студентов ФПМИ, но может быть полезен и для студентов других специальностей.

Ил. 65

Работа подготовлена двумя членами группы ПМ-13.

От авторов:

Ильиных Данил Сергеевич aka Woodroof:

Данный конспект составлен по лекциям, прочитанным В. М. Стасышиным в феврале-мае 2004 года. В него также включены фрагменты лекций и примеры программ, выложенные на сайте http://ami.nstu.ru/~vms. Глава «Файловые системы» взята из лекций Н. Л. Долозова, прочитанных год назад и набранных нами с Вороновым Константином.

Хочется сказать спасибо В. М. Стасышину за хорошие лекции, Воронову Константину за набранные лекции по сетевому взаимодействию и средствам RPC, Логвиненко Александре за помощь при подготовке данного документа.

При обнаружении ошибок пишите: woodroof@ngs.ru

Воронов Константин Игоревич aka MOV_ah:

Моя работа заключалась в отыскании и исправлении ошибок и неточностей, допущенных Woodroof'ом при подготовке электронной версии, а также в дополнении исходной работы.

Хочу поблагодарить всех тех, кто морально поддерживал нас при выполнении этой работы (хотя таких не было), Ильиных Данила за выполнение большей части работы по подготовке материала к данным лекциям и Владимира Михайловича Стасышина за столь сжатое и простое изложение столь сложных и объёмных понятий управления ресурсами в операционных системах, неизбежно встречающихся при углублённом изучении операционных систем типа UNIX или MS-DOS.

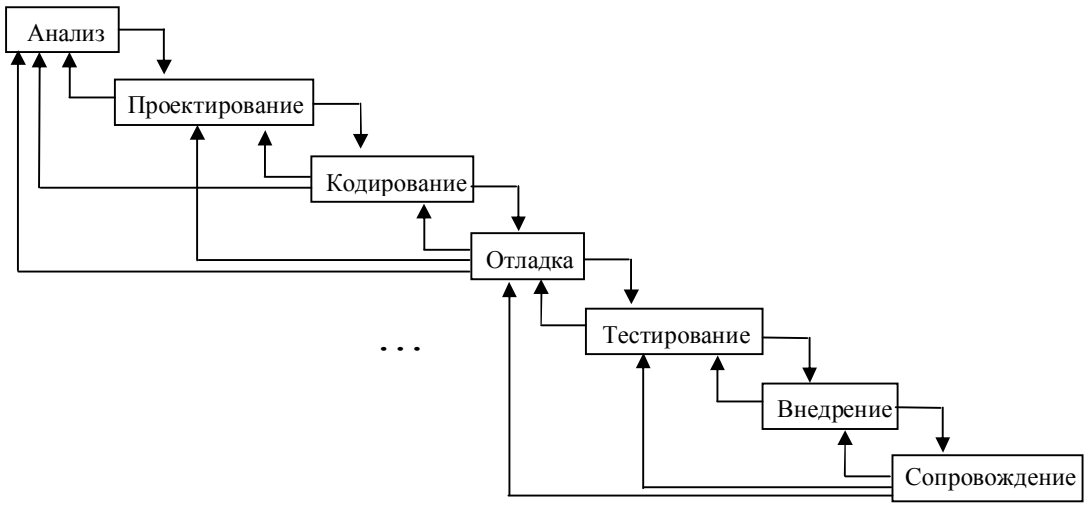
По всем вопросам обращайтесь: admin@ztx.ru

Жизненный цикл программы.

Что понимается под разработкой программ? Под разработкой программы понимается технологический многостадийный процесс создания отторгаемого программного продукта, на разных стадиях разработки которого работают разные люди.

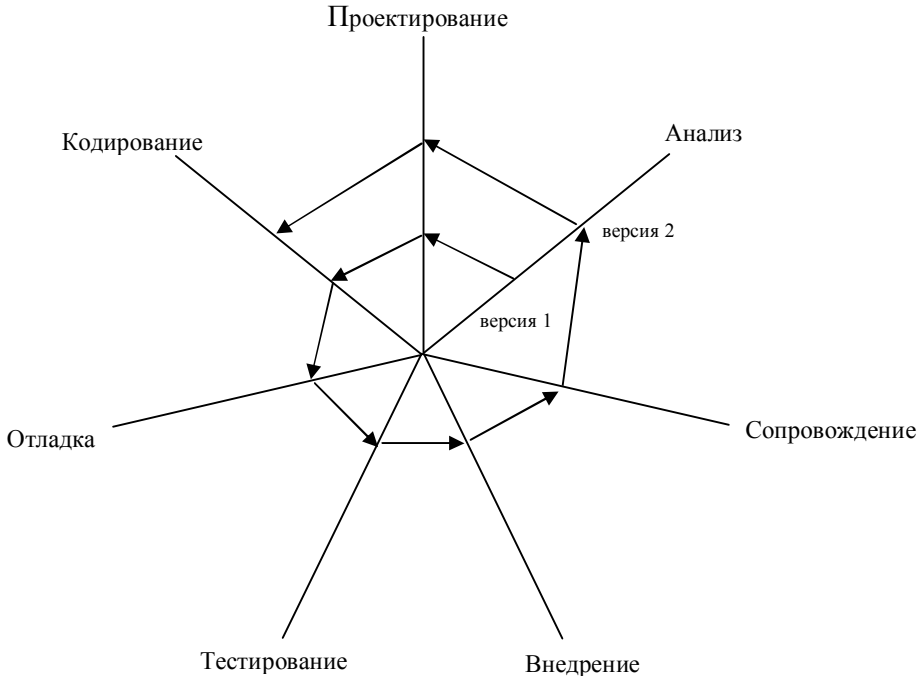
Существуют две основные модели жизненного цикла программы:

Каскадная:



В любой момент разработки возможен возврат назад на любой этап.

Спиральная:



Анализ: На этом этапе нужно решить, стоит ли создавать новую программу. Возможно, уже созданы программы, решающие данную задачу. Если есть средства разрешения проблемы, которые будут заведомо хуже вашего продукта, то и в этом случае нужно проанализировать, будут ли затраты на создание программного продукта соизмеримы с потерями при использовании худших средств решения проблемы. И только проанализировав затраты, нужно приступать (или не приступать) к созданию программного продукта.

Проектирование: Нужно запомнить одну непреложную истину: нельзя сразу приступать непосредственно к процессу кодирования. Такой подход применим только для очень маленьких программ, но никак не к программным продуктам.

На этом этапе следует провести тщательный анализ исходной задачи, построить различные схемы и диаграммы. Нужно продумать несколько подходов к решению проблемы и выбрать из них наилучший. Необходимо выделить относительно независимые подзадачи, отделить их друг от друга в разных модулях, спроектировать потоки данных между модулями и внутри их.

Кодирование: Существуют различные стили программирования. Среди них нет лучшего или худшего, все они имеют право на жизнь, впрочем, стоит различать свойственные разным языкам стили программирования и не писать на языке Си так, как принято на Фортране.

Тем не менее, есть некоторые общие аспекты в написании программ, которых нужно придерживаться:

Программа должна быть ясной. Не так сложно несколько раз нажать кнопку Tab, чтобы отделить вложенные блоки от внешних. Желательно разделять относительно независимые блоки одной или несколькими пустыми строками, это увеличивает читабельность программы.

Разумное использование комментариев и согласованное употребление отступов может сделать чтение и понимание программы более приятным занятием. При неправильном использовании комментариев может серьезно пострадать читабельность программы.

К сожалению, компилятор не может проверить, что комментарий:

- содержателен;
- имеет какое-то отношение к программе;
- не устарел.

В большинстве программ можно найти труднопонимаемые, противоречивые и просто неверные комментарии. Плохой комментарий хуже его отсутствия.

Если что-то может быть выражено *непосредственно конструкциями языка*, так оно и должно быть сделано, простого упоминания в комментарии недостаточно:

```
//переменную "v" надо проинициализировать
//переменная "v" должна использоваться только функцией "f()"
//перед вызовом любой другой функции из этого файла, вызовите функцию "init()"
//вызовите функцию "cleanup()" в конце вашей программы
```

При правильном использовании языка программирования, такие комментарии часто становятся ненужными.

Если что-либо ясно выражено в языке, не надо это повторять второй раз в комментарии:

```
a=b+c;          //a принимает значение, равное b+c
count++;        //увеличили счётчик
```

Хорошим стилем будет использование комментариев в следующих случаях:

- Комментарии в начале каждого файла исходного кода, где поясняются основные объявления, делаются ссылки на литературу и приводятся наиболее важные соображения по поводу сопровождения и т. п.
- Комментарии для каждого класса, шаблона и пространства имён.
- Комментарий для каждой нетривиальной функции, в котором указано её назначение, использованный алгоритм (если он не очевиден) и, может быть, предположения, которые она делает об окружении.
- Комментарии для каждой глобальной переменной, переменной из пространства имён и константы.
- Небольшие комментарии в тех местах, где код неочевиден и/или непереносим.
- Очень редко в других случаях.

Удачно подобранный и написанный набор комментариев является существенной частью хорошей программы. Написание «правильных» комментариев может оказаться не менее сложной задачей, чем написание самой программы. Стоит развивать это искусство.

Если в функции используется только //-стиль комментариев, то любой фрагмент функции может быть «закомментирован» (временно исключён) при помощи /* */ и наоборот, но не стоит использовать оба типа комментариев как для исключения временно ненужных частей кода, так и для пояснения текста программы.

Стоит давать осмысленные имена переменным, как count, day, PersonName. Их использование не позволит вам запутаться, тогда как переменные j72, Masha154, qwerty, aaa могут причинить кучу неприятностей.

- Эффективность стоит повышать в конце работы, когда вы уже добьетесь правильной (пусть неоптимальной) работы программы. После можно выделить критические части программы и, возможно, переписать его средствами Си низкого уровня, а то и вовсе на Ассемблере. Пусть Вы переделаете половину программы, но Вы будете знать, что ваша программа работает, иначе это может привести к бесконечной оптимизации, путанице и стопору в работе.

Отладка: Программа должна работать на любых входных данных. Если они неправильны – выдавать сообщение об ошибке. Нельзя надеяться на правильность входных данных, и, тем более, на правильную последовательность действий пользователя. Пользователь по природе своей склонен нажимать все кнопки и в разной последовательности. Он из любопытства своего будет мучить вашу программу во всевозможных ситуациях, и в этом случае надо не потерять лица, кто знает, быть может, пользователь будет пользоваться продукцией вашего конкурента, который будет следить за неправильными данными и неправильными действиями пользователя.

Тестирование: К этому этапу создания программного продукта стоит привлекать посторонних людей. Лучше, если они не будут знать, как работает ваша программа. Мыслительные процессы других людей отличаются от ваших; велика вероятность, что они найдут ошибки, которые Вы не заметили.

Главный тезис: «Все ошибки предпоследние».

Внедрение: Обычно ограничиваются вышеперечисленными этапами. Принято считать, что главное – написание программы, а пользоваться ею начнут автоматически. Увы, все несовершенно в этом мире, и потому наши мечты не склонны сбываться. Приходится объяснять рядовому пользователю, что ваша программа гораздо лучше программы конкурента, работает раза в два быстрее, да и вообще… И тут разработчики сталкиваются с извечной консервативностью конечных пользователей. Ну как объяснишь обычному человеку, что пингвины гораздо лучше разноцветных окон?!?

Этап внедрения зачастую отнимает больше сил и времени, чем все остальные этапы разработки программного продукта, и не нужно этим этапом пренебрегать.

Сопровождение: Этап сопровождения заключается в поддержке ранее выпущенных программ, исправлении обнаруженных ошибок, выпуске новых версий… Чем лучше Вы все сделали раньше, тем легче будет этот этап. Но если хоть на одном из этапов разработке программного продукта Вы поленились, то процесс сопровождения может превратиться в сущий Ад.

Основные понятия операционной системы (ОС).

Все работы организуются как взаимодействие отдельных процессов.

Процесс – единица потребления ресурсов.

Процесс – программа, выполняемая в своём виртуальном пространстве.

Процесс – последовательность операций программы на этапе её исполнения (программа = исполняемый код).

Каждой программе в любой момент времени может соответствовать один или несколько процессов или не соответствовать ни одного. Процесс – последовательность операций, никакие асинхронные действия в рамках процесса не допустимы. Распределение ресурсов между процессами выполняется резидентной частью операционной системы, которая носит название *ядра*.

Ядро состоит из двух секций:

- секция управляющих структур (системные таблицы);
- программная секция:
  а. машинно-независимая часть (написана на Си, по разным оценкам примерно 90-92% кода);
  б. машинно-зависимая часть (смесь Си и Ассемблера)

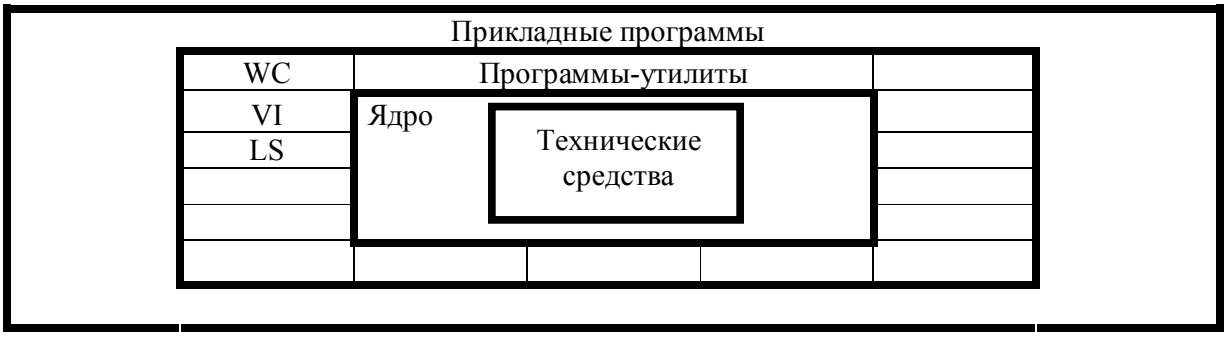
Ядро управляет всеми ресурсами.

Функции ядра:

- Инициализация системы.
- Управление процессами: их создание, синхронизация, диспетчеризация и взаимодействие процессов между собой.
- Планирование очередности предоставления процессам времени центрального процессора (ЦП) по некоторому алгоритму в режиме разделения (квантования) времени (диспетчеризация).
- Выделение выполняемому процессу оперативной памяти и управление свопингом.
- Выделение внешней памяти с целью обеспечения эффективного хранения и выборки информации.
- Доступ процессов к периферийным устройствам (диски, терминалы, сетевое оборудование).
- Приём от процессов запросов на обслуживание.

Ядро не занимается обслуживанием пользователей, для этого существует специальный аппарат - набор системных программ – утилит, выступающих в качестве посредника между ядром и пользователем.

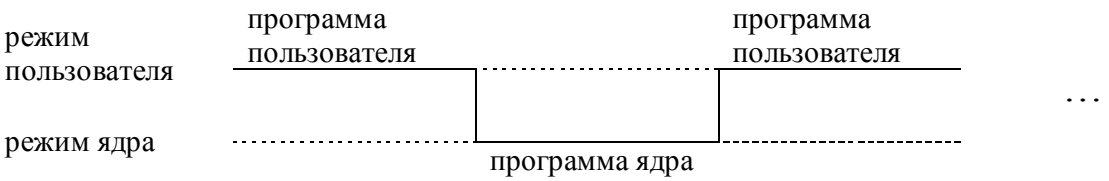
Ядро становится относительно компактным (по различным оценкам 10-15 тыс. строк на языке Си, 2-3 тыс. строк на языке Ассемблер). Учитывая, что ядро можно конфигурировать и компилировать «под себя», а часть функций можно выделить в подгружаемые модули, то можно добиться значительного уменьшения размера ядра.



Выполнение пользовательских процессов в ОС UNIX осуществляется на двух уровнях:

- *уровень пользователя* (пользовательский режим, режим задачи, пользовательская фаза);
- *уровень ядра* (системный режим, режим ядра, системная фаза).

Когда процесс обращается к ОС для получения некоторого сервиса, режим выполнения процесса переключается с режима задачи на режим ядра. Смена режима не связана с порождением нового процесса, а является лишь иным состоянием исходного процесса.

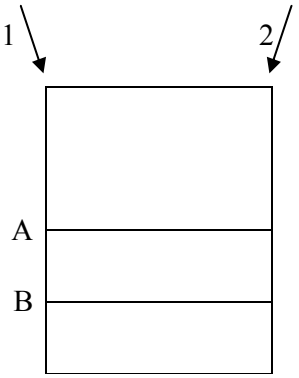


Даже если пользовательские процессы не используют системные вызовы, ядро все равно каждый раз запускает режим ядра от имени пользователя.

Отличия этих режимов:

1. В режиме задачи процессы имеют доступ только к своим собственным командам и к своим данным, и не имеют доступа к командам и данным ядра и других процессов. В режиме ядра процессам доступны адресные пространства как ядра, так и всех пользовательских процессов.
2. В пользовательском режиме не могут выполняться некоторые привилегированные команды, доступные только в режиме ядра (например, управление регистрами процессора).
3. Процессу, функционирующему в системной фазе, соответствует реентерабельный (re|enter|able – программа, допускающая повторный вход) код ядра, в пользовательской фазе процессу соответствует пользовательский код программы.

Что такое реентерабельные программы? Рассмотрим на примере:



Пусть процессы 1 и 2 – два процесса одной и той же программы. Процесс 1 выполняется до точки A, после чего система передаёт управление процессу 2, который доходит до точки B. После того, как управление вернётся процессу 1, он должен продолжить своё выполнение с момента A. Кроме того, если процесс 1 изменяет какие-либо данные, то это никак не должно отразиться на работе процесса 2.

В связи с особенностями реентерабельных программ, возникают некоторые ограничения:

- программа не модифицирует себя, свои данные;
- никакие данные внутри программы не используются;
- все нужные данные программа запрашивает динамически;
- каждый вариант программы работает со своими отдельными данными.

Пример на переключение режимов:

Интерпретатор команд читает входной поток данных с терминала. Интерпретатор (shell) в режиме задачи (task) посылает запрос к ОС для работы с данным устройством. От имени Shell в режиме ядра (kern) запускается процесс, работающий с устройством (в данном случае – с клавиатурой), выполняющий чтение символов. Как только ввод заканчивается, процесс, отвечающий за ввод, завершается и Shell снова переходит в режим задачи, а затем интерпретирует команды (распознаёт входную последовательность) и осуществляет вызов некоторой программы в соответствии с командой пользователя.

Данный процесс можно проиллюстрировать схемой:

Shell (task) → запрос к ОС → от имени Shell (kern) запускается процесс, работающий с клавиатурой (чтение символов) → Shell (task) – интерпретация команд → вызов некоторой программы

Нормальное состояние любого пользовательского процесса (в том числе и Shell) – режим задачи (task). Для работы с устройствами используется режим ядра (kern).

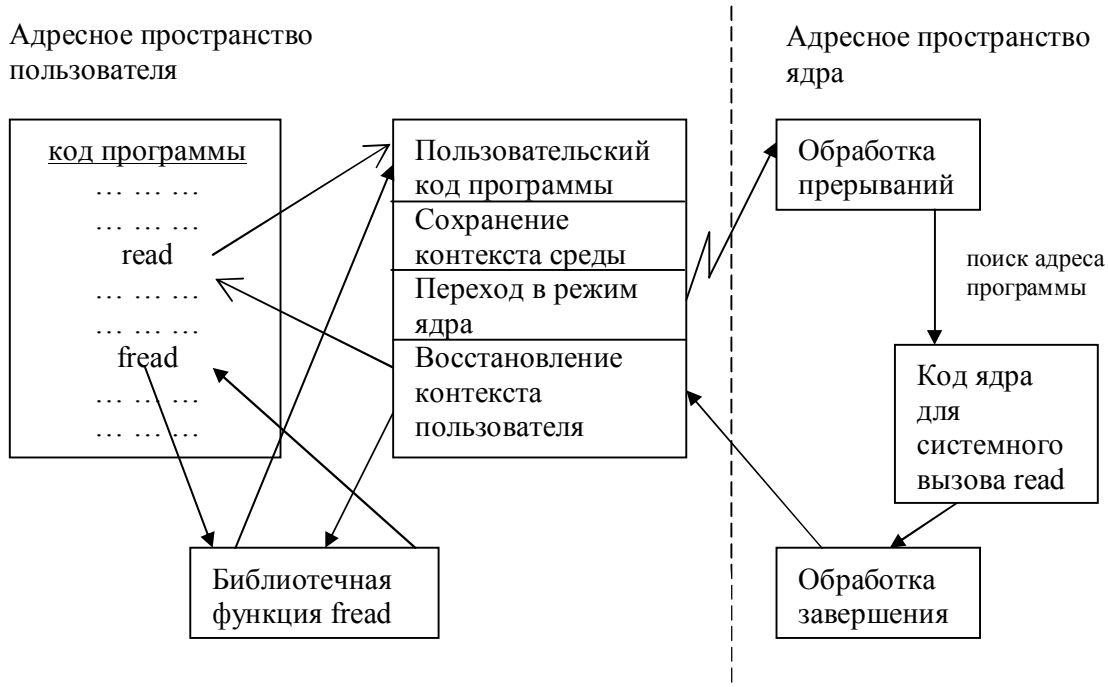
Поскольку система UNIX – система разделения времени, и основной её задачей является поддержание множества программ, находящихся на различных стадиях выполнения, то предоставление услуг ядра этим программам должно выполняться наиболее безопасным образом. Таким безопасным механизмом является механизм системных вызовов.

Механизм системных вызовов – то, посредством чего можно запросить некоторые ресурсы ядра. В любом случае, как бы механизм обработки вызовов к ядру не назывался, он выливается в обработку прерываний.

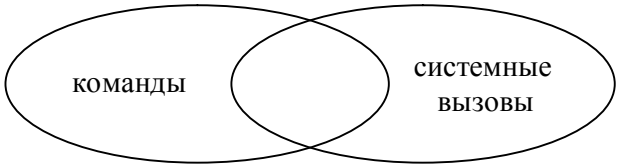
Рассмотрим две функции: библиотечную и системный вызов.

1. nread = fread (inputbuf, OBJSIZE, num, filter) – библиотечная функция;
2. nread = read (field, inputbuf, BUFSIZE) – системный вызов.

Механизм работы этих функций проиллюстрируем схемой:



Как правило, командам shell соответствуют системные вызовы. Но есть команды без системных вызовов и системные вызовы, которым не соответствуют команды.



В системе UNIX одновременно выполняется множество процессов, при этом некоторые из них могут одновременно выполняться в режиме ядра. Возможен случай, что некоторые процессы будет одновременно работать с системными таблицами. Некоторые участки системных фаз являются критическими в том смысле, что пока один процесс не вышел из критического участка, другой процесс не может в него войти. Данный механизм используется для обеспечения целостности ядра.

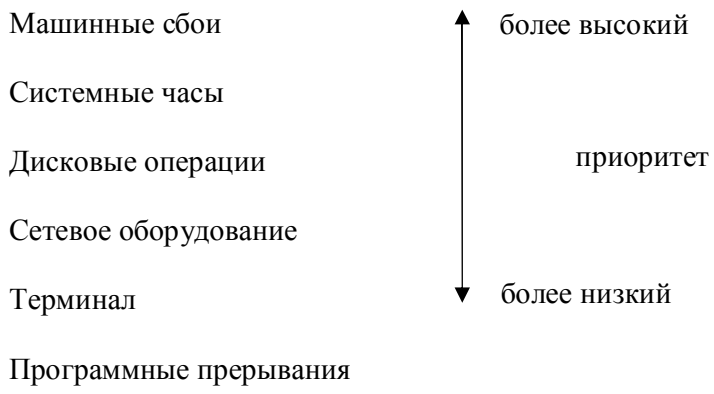
Для синхронизации процессов в системной фазе используется аппарат событий, который доступен только в системной фазе и не доступен в режиме пользователя.

В ОС UNIX используется простейший вариант механизма аппарата событий, когда процесс, приостановленный от лица некоторого другого процесса, размораживается только этим процессом.

Система UNIX позволяет ряду внешних устройств асинхронно прерывать работу центрального процессора (ЦП) в соответствии с жестко заданной системой приоритетов.

Общая схема: По получении сигнала прерывания, ядро ОС сохраняет свой контекст, определяет причину прерывания, обрабатывает это прерывание, после чего восстанавливает свой старый контекст. В процессе обработки прерываний ядро учитывает приоритет прерываний и при их обработке блокирует прерывания более низкого уровня.

Система прерываний следующая:



Мультипрограммирование и свопинг.

Мультипрограммирование складывается из последовательности действий, при реализации которых решается вопрос, какой пользовательский процесс должен выполняться. В любой момент времени только один пользовательский процесс активен, все остальные приостановлены.

Множество приостановленных процессов делится на следующие группы:

1. процессы, готовые к выполнению, лишь ожидающие предоставления им ресурсов центрального процессора;
2. блокированные процессы, ожидающие появления некоторого события, только после которого они могут продолжить свою работу.

Режим квантования времени реализуется следующим образом:

1. системные часы, которые с определенной дискретностью генерируют определённые прерывания;
2. системные вызовы.

В обоих случаях генерируются прерывания, управление получает диспетчерский процесс, выполняется переоценка приоритетов, и, как следствие этого, выполняется смена активного процесса.

Пока мы рассматривали идеальный случай – все процессы помещаются в оперативную память, реально же объём оперативной памяти ограничен, и поэтому возможна ситуация, когда отдельные образы процессов могут в неё не помещаться.

Часть приостановленных процессов хранится во внешней памяти; операция перемещения образа процесса из оперативной памяти во внешнюю память и обратно носит название *свопинга*.

Таким образом, функция мультипрограммирования двоякая:

1. управление процессом предоставления ресурсов ЦП;
2. управление свопингом.

Для активизации процесса диспетчерский процесс оперирует понятием приоритета. Приоритет является функцией от времени загрузки процесса в оперативную память. При этом, чем дольше процесс находится в активном состоянии, тем ниже его приоритет. Если процесс сориентирован на вычисления и может полностью занимать ресурсы ЦП, его приоритет понижается. Если процесс занимается обращением к внешней памяти, используя системные вызовы, его приоритет не понижается. В промежутке бездействия ЦП могут занимать другие процессы.

Пользователь имеет ограниченную возможность влиять на величину приоритета, причём обычный пользователь может только понижать приоритет, а пользователь с правами *суперпользователя* (root), может также и повышать приоритет.

Свопингом также занимается диспетчерский процесс. Необходимость в свопинге возникает, когда нужно активизировать процесс, не находящийся в оперативной памяти. Если процесс, подлежащий загрузке в оперативную память, найден, то проверяется, достаточно ли места для его размещения в оперативной памяти.

Если места достаточно, процесс загружается и становится кандидатом в борьбе за ресурсы ЦП. Место, занимаемое процессом в области свопинга, освобождается.

Если места в оперативной памяти недостаточно, решается вопрос о выгрузке образа процесса во внешнюю память. Прежде всего, определяется процесс-кандидат, подлежащий выгрузке, из числа процессов, находящихся в системной фазе. Из этих процессов выбирается процесс, занимающий в оперативной памяти наибольшее количество пространства. Если в системной фазе процессов нет, выгружается процесс, наиболее долго находящийся в оперативной памяти. Если в области свопинга находится несколько процессов в готовом состоянии, то кандидатом на загрузку является процесс, который наиболее долго находится во внешней памяти.

Процессы, запущенные от имени ядра, не подлежат свопингу – ядро не выгружаемо.

Управление памятью. Общие замечания.

Механизм виртуальной памяти.

Каждый процесс работает в своём *виртуальном адресном пространстве*. Эта идея позволяет добиться следующего:

1. Обеспечивает иллюзию практически неограниченной пользовательской памяти при наличии оперативной памяти существенно меньших размеров.
2. Обеспечение защиты одной пользовательской программы от другой.
3. Позволяет предоставлять ОС возможность динамически перераспределять оперативную память между одновременно выполняющимися пользовательскими процессами.

Средства обеспечения виртуальной памяти:

1. Машинные команды в образе процесса можно выделять в отдельный процедурный сегмент с тем, чтобы можно было работать с программой как с реентерабельной.
2. Если программа большая, её можно разбить на части и выполнять либо последовательно в рамках одного процесса, либо параллельно как независимые процессы.
3. Наличие массивов может существенно увеличить размер программ. В ОС UNIX существует возможность трактовать массивы данных как файлы, тем самым исключая их из виртуального адресного пространства.

   Отрицательной чертой такого подхода является увеличение времени обращения к данным.

   Компенсационные методы для третьего средства обеспечения:

   - Использование обмена с блочными устройствами через буферизируемую кэш-память. При чтении элемента данных из массива, трактуемого как файл, читается целиком весь блок, в котором содержится данный элемент. Весь блок помещается в кэш-память. Впоследствии, при чтении следующего элемента того же массива, предварительно проверяется его наличие в кэш-памяти.

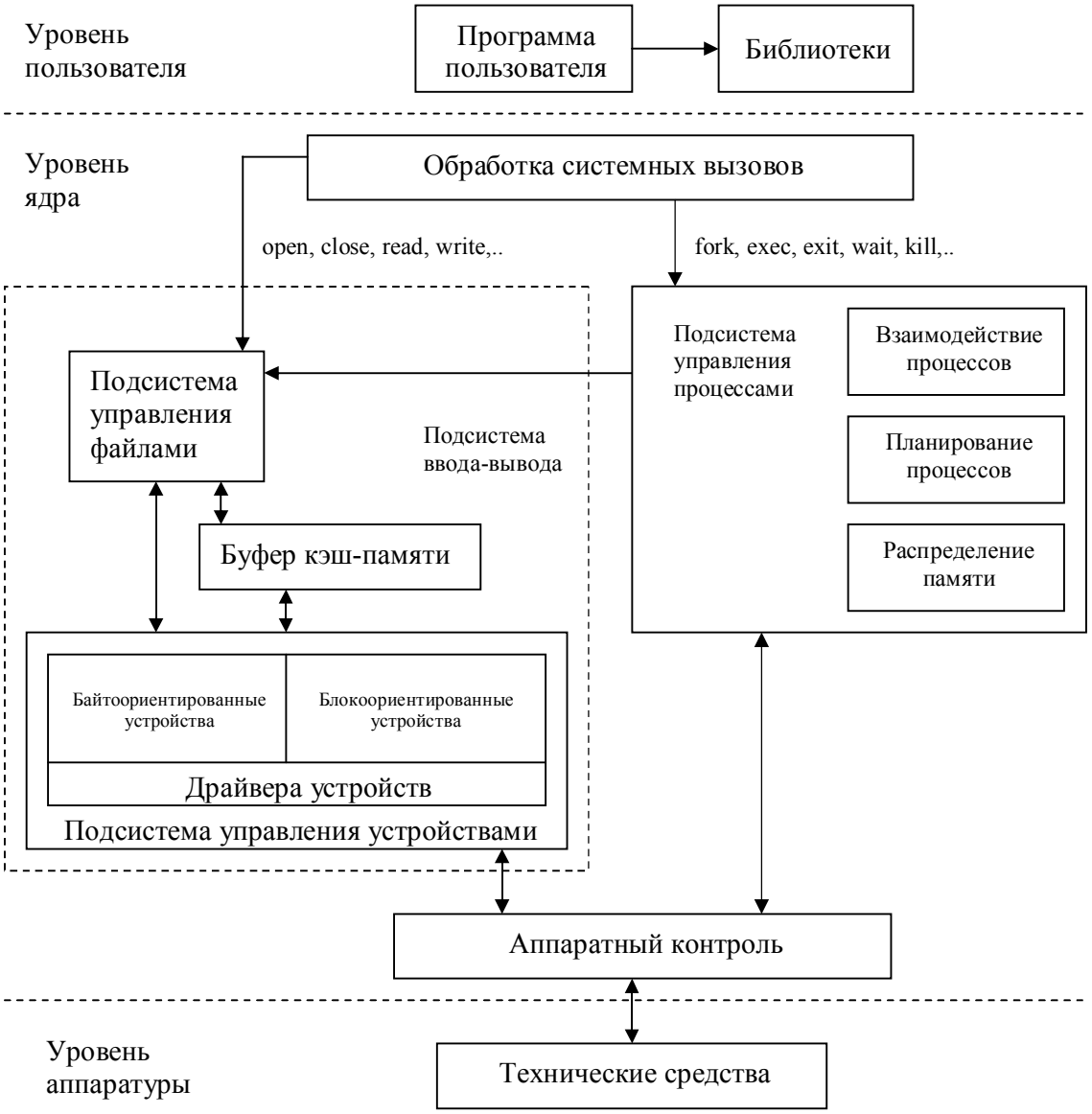
   - Использование файла данных как одномерного массива с прямым доступом. Таким образом, достигается увеличение скорости доступа за счёт максимального упрощения структуры файла.

Архитектура ОС.

Три основные компоненты:

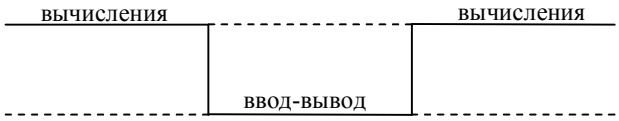
1. Подсистема управления файлами.
   Управляет файлами, свободным дисковым пространством. Размещает записи файлов, выполняет поиск данных в файле.
2. Подсистема управления процессами.
   Резервирует ресурсы, определяет последовательность выполнения процессов, их синхронизацию и взаимодействие. Управляет оперативной памятью, реализует механизм разделения времени.
3. Подсистема управления устройствами.

Первые две из них являются машинно-независимыми, последняя компонента является частично машинно-зависимой. Подсистемы управления процессами и устройствами в совокупности образуют подсистему ввода-вывода.

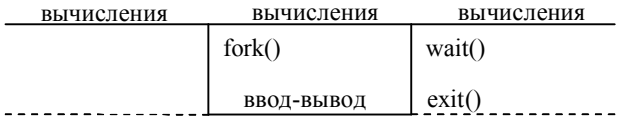


С практической точки зрения процесс является объектом, созданным в результате выполнения системного вызова fork(). Тот процесс, который создаёт, называется родительским процессом; тот, который создаётся, называется порождённым процессом.

Процесс предусматривает строго последовательные действия. При необходимости выполнить действия асинхронно с процессом, необходимо породить новый процесс.



Если порождается новый процесс:



В тот момент, когда возникает потребность в операциях ввода-вывода, создается новый процесс, после чего процессы как-либо синхронизируются.

Загружаемый ядром процесс может состоять из трёх сегментов (областей):
- процедурный (сегмент текста);
- сегмент данных;
- сегмент стека.

Сегмент текста содержит команды и константы программы, которые составляют процесс. Сегмент данных содержит те данные, которые определены на этапе компиляции исходного текста программы (bss-данные). Сегмент текста и сегмент данных в совокупности образуют *образ процесса*.

Процедурный сегмент может быть разделяемым с тем, чтобы сделать программу реентерабельной.

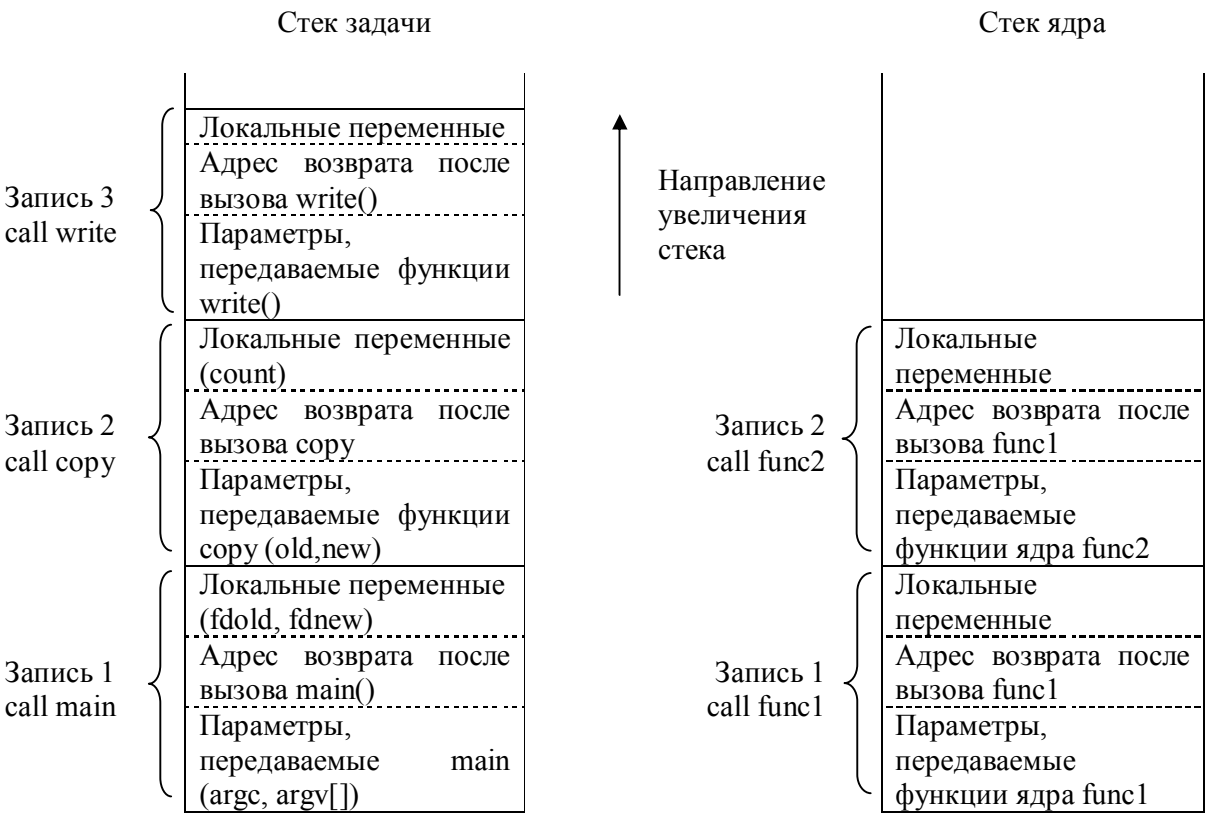
Сегмент стека создаётся автоматически во время запуска процесса. Размер определяется динамически. Стек состоит из логических записей активации, которые помещаются в стек при вызове очередной функций и выталкиваются из стека при возврате из функций.

Запись активации содержит параметры вызываемой функции, локальные переменные этой функции и данные, необходимые для восстановления предыдущей записи активации.

Поскольку имеется два режима: пользовательский и ядра, для каждого из этих режимов существует свой отдельный стек.

Пример:

```
#include <fcntl.h>
char buffer[2048]; //bss-данные
void copy (int, int);
int main (int argc, char* argv[])
{
    int fdold, fdnew;
    ……
    fdold = open (argv[1], O_RDONLY);  /*Исходный файл открыт для чтения*/
    fdnew = creat (argv[2], 0666);          /*Создан файл с определенными правами*/
    copy (fdold, fdnew);
    exit(0);
}
void copy (int old, int new)
{
    int count;
    while ((count = read (old, buffer, sizeof (buffer))) > 0)
            write (new, buffer, count); ⇐ Рассмотрим стек в этой точке
}
```



Предполагаем, что write вызывает функцию func1, которая в свою очередь, вызывает func2.

При возврате из функций записи «выталкиваются» из стека.

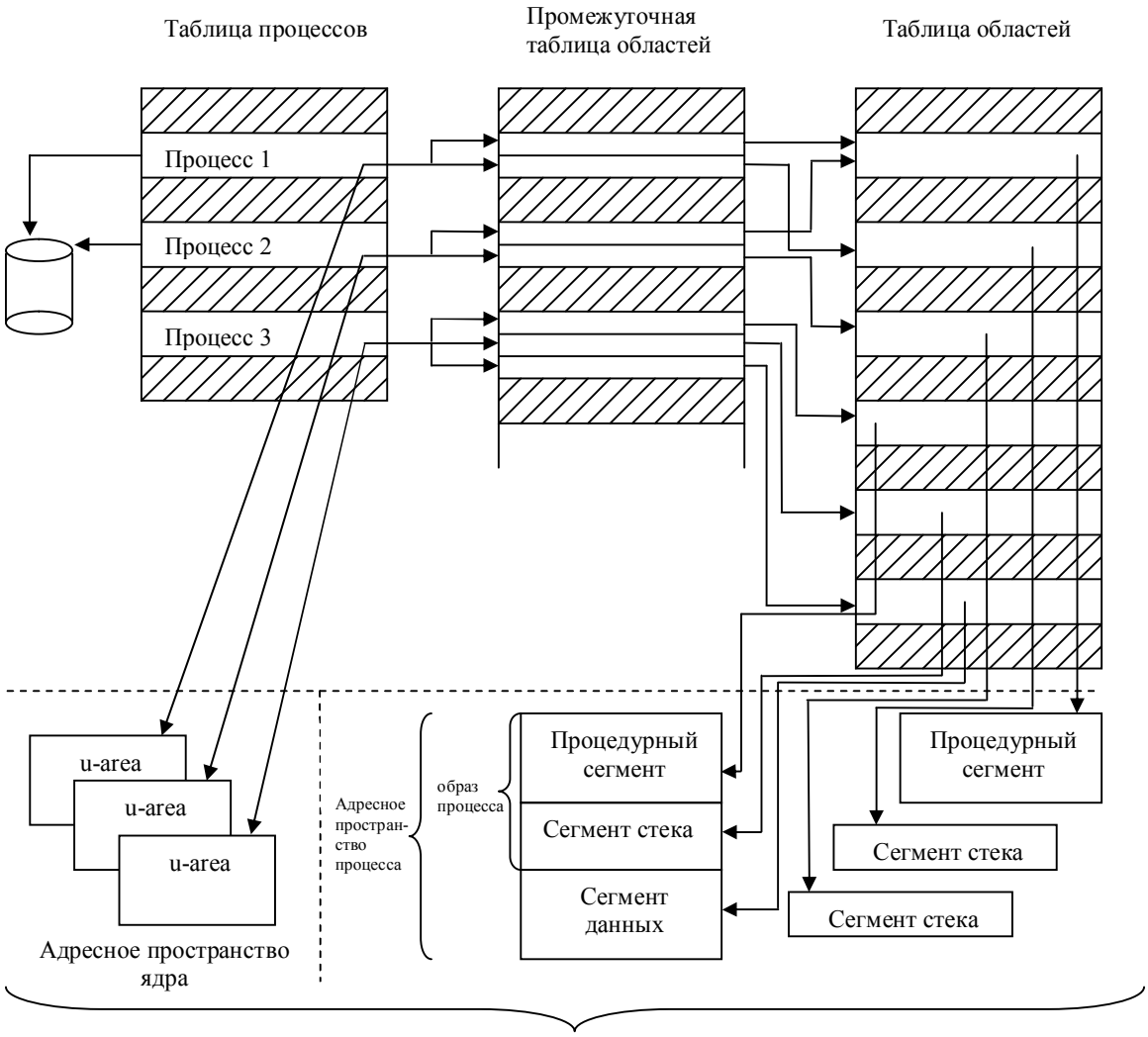
В общем случае, области данных может и не быть.

Каждому процессу соответствует точка входа в таблицу процессов ядра и выделяется область оперативной памяти, отведенная под задачу пользователя (u-area – u-область). Таблица процессов включает в себя указатели на промежуточную таблицу областей процессов, элементы которой являются указателями собственно на таблицу областей.

Областью называется непрерывная зона адресного пространства, которая выделяется для размещения текста, данных и стека.

Записи в таблице областей описывают атрибуты этой области и указывают, где конкретно размещена в памяти эта область.

Уровень косвенной адресации посредством промежуточной таблицы областей позволяет независимым процессам совместно использовать ту или иную область.



Процессы 1 и 2 используют один и тот же процедурный сегмент. Порождаемый процесс создаётся копированием всех записей в таблицах.

Контекст процесса. Информационное взаимодействие процесса.

Контекстом процесса является его состояние, определяемое текстом, значениями глобальных переменных, значениями используемых машинных регистров, состоянием таблицы процессов и связанной с ней u-области, значениями из таблицы областей, содержимым стеков задачи и ядра, связанные с этим процессом.

Контекст распадается на:
- пользовательский контекст;
- регистровый контекст;
- контекст системного уровня.

Пользовательский контекст определяется содержимым сегмента программного кода, сегмента данных, стека задачи и содержимым разделяемых областей памяти.

Регистровый контекст определяется содержимым аппаратных регистров, связанных с этим процессом.

Контекст системного уровня подразделяется на:
- статическую часть;
- переменное число динамических частей.

Статическая часть содержит:

1. Элемент таблицы процессов:

Состояние процесса, физический адрес в основной или внешней памяти u-области, идентификатор пользователя, от имени которого запущен процесс, идентификатор процесса, идентификатор события (в случае, если процесс приостановлен), ссылка на промежуточную таблицу областей…

2. u-область – индивидуальная для каждого процесса область в пространстве ядра, обладающая тем свойством, что виртуальные адреса u-областей всех процессов одинаковы.

u-область содержит:
- указатель на элемент таблицы процессов;
- счётчик времени, в течение которого процесс занимал процессор в режиме ядра и в режиме задачи;
- параметры и результаты системных вызовов;
- таблица дескрипторов открытых файлов;
- коды ошибок (если таковые есть)…

Динамическая часть:

Один или несколько стеков, которые используются процессом при его выполнении в режиме ядра.

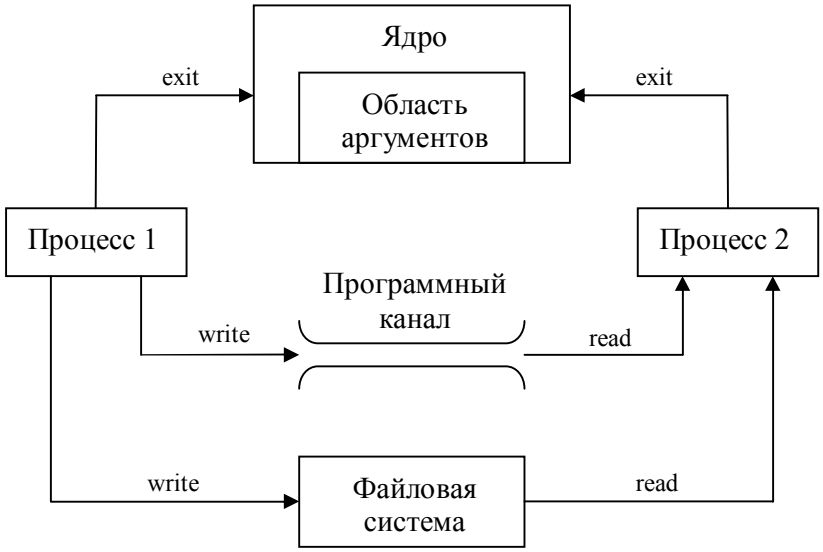
Число стеков определяется числом уровней прерываний.

Информационное взаимодействие.

Существуют три способа информационного взаимодействия между процессами:
1. Путём передачи внешних параметров.
2. Через файловую систему.
3. Через программный канал.

Логическое взаимодействие.

Выполняется только через ядро системы с использованием механизма *сигналов*.



Через программный канал могут взаимодействовать только родственные процессы, через файловую систему – любые. Логическое взаимодействие осуществляется только через ядро.

Состояние процесса.

Процесс (на самом верхнем уровне) может находиться в четырёх состояниях:

- процесс активен в состоянии задачи;
- процесс активен в состоянии ядра;
- процесс заблокирован (находится в состоянии сна);
- процесс находится в состоянии готовности.

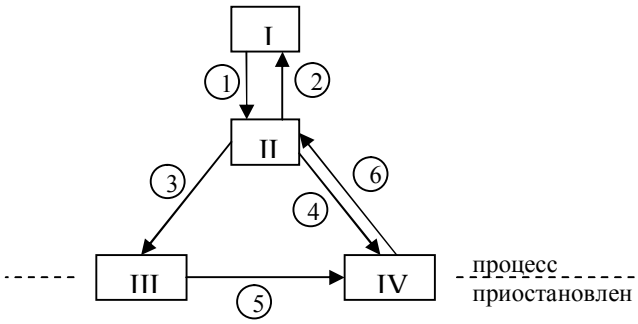
В отличие от первого и второго состояний, в третьем и четвёртом может находиться множество процессов одновременно.

Процесс активен, если процессор выполняет команды этого процесса. В каждый момент времени активным может быть только один процесс.

Процесс заблокирован, если он ожидает некоторого условия или сигнала, до наступления которого он не может выйти из этого состояния.

Процесс находится в состоянии готовности, если выполнены все условия для перехода в активное состояние и процесс ожидает предоставления процессора.

Процесс может переходить из одного состояния в другое.



1 – обращение к ядру ОС или прерывание.

2 – возврат после обработки прерывания или после получения услуг ОС.

3 – процесс должен ожидать некоторые события (например, завершения операции ввода/вывода).

4 – завершение кванта времени, выделенного процессу.

5 – «пробуждение» процесса при наступлении некоторого события (например, завершение операции ввода/вывода, но ЦП при этом занят).

6 – выделение процессу процессора.

Подсистема управления файлами.

Термин «файловая система»

- иерархия каталогов
- часть ядра ОС, которая управляет каталогами и файлами

В мире ОС UNIX термин «файловая система» оказался перегруженным.

Типы файловых систем

локальная файловая система

- S5, UFS — «классические» ФС для UNIX
- proc
- ext2 (Linux)
- …

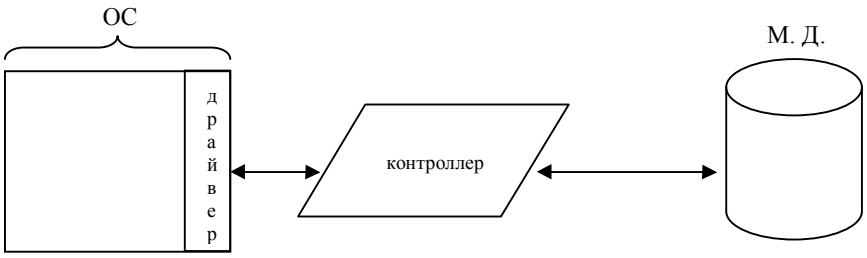
иерархия каталогов и файлов на локальном устройстве

распределённая файловая система

- NFS (Sun Microsystems)
- RFS (AT&T)
- …

может находиться не только на локальном устройстве, но и на удалённых устройствах

Физическая модель магнитного диска.

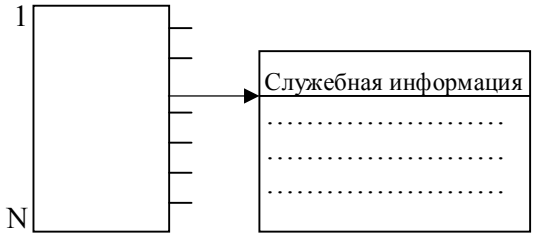


В качестве основного запоминающего устройства в ОС UNIX используется жёсткие диски. Связь с ними осуществляется через дисковый контроллер (электрическая плата, присоединённая к шине периферийных устройств). Контроллер управляет операциями низкого уровня (пересылка данных, ошибки, чтение/запись).

Термины:

- сектор (512 байт)
- трек
- дорожка
- цилиндр
- физический адрес (№ цилиндра, № дорожки в цилиндре, № сектора)

Логическая модель магнитного диска.



Главная загрузочная

запись MBR (boot-сектор)

1. Распределение ресурсов.

Единица распределения ресурсов кратна размеру сектора (В UNIX'е – блок, в DOS'е - кластер).

2. Состояние единицы распределения ресурса (занято, свободно…).

Организация файловых систем S5 и UFS («классические» ФС).

Замечание.

Обозначение дисков (Linux):

/dev/hda – первый диск

/dev/hdb – …

/dev/hdc – …

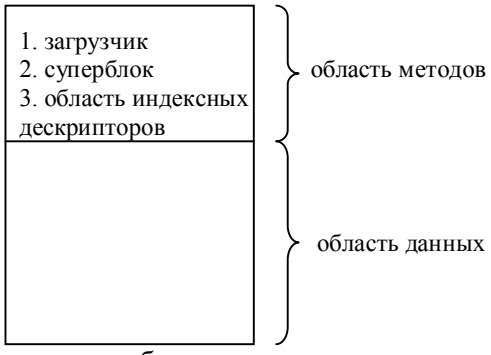
Разделы:

hdx1
…
hdx4
} используются только первичные разделы

Логические разделы нумеруются, начинается с 5.

S5.

Расположение ФС S5 в разделе диска может быть проиллюстрировано:



Загрузчик – область в разделе, используемая для загрузки ОС (находится в разделе, где создаётся корневая файловая система, но место под него резервируется всегда).

Суперблок – содержит самую общую информацию о ФС (размер ФС, размер области индексных дескрипторов, их число, список свободных блоков, свободные индексные дескрипторы и т. д.). Суперблок всегда находится в оперативной памяти.

В суперблоке находятся:

1. Счетчик числа свободных блоков. Переменная s_nfree.
2. Список свободных блоков. Массив s_free[].
3. Счетчик числа описателей файлов. Переменная s_ninode.
4. Список описателей файлов. Массив s_inode[].

Область индексных дескрипторов состоит из inode'ов. С каждым файлом связан один inode. В inode хранится вся информация о файле, кроме его имени.

Область данных:

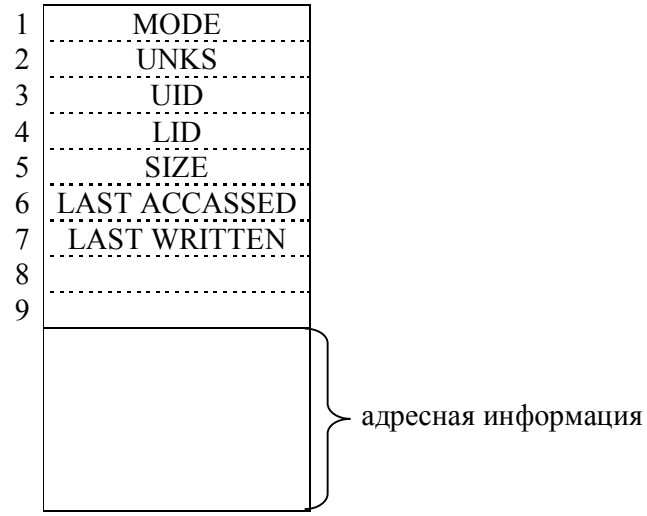
Там расположены как обычные файлы, так и файлы каталогов (в том числе корневой каталог).

Специальные файлы представлены в ФС только записями в соответствующих каталогах и индексными дескрипторами специального формата, т. е. места в области памяти не занимают.

Структура индексного дескриптора (inode).

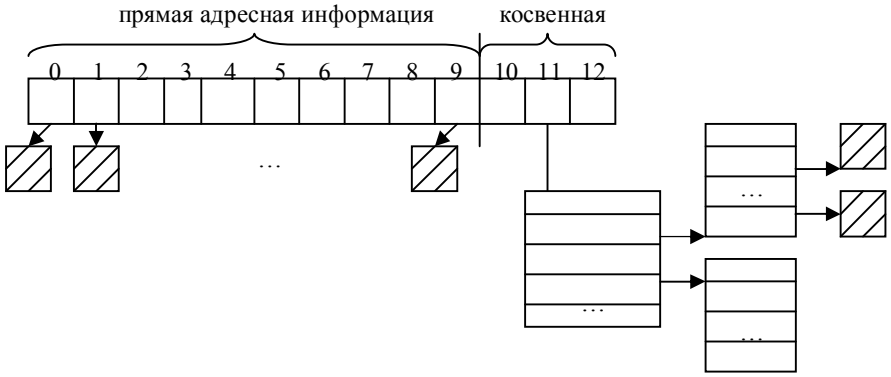
Вся информация о файлах, кроме их содержимого и имени, находится в так называемых описателях файлов. Каждому файлу соответствует один описатель. Описатель имеет фиксированный формат и располагается непрерывным массивом, начиная со второго блока. Общее число описателей (максимальное число файлов) задаётся в момент создания ФС. Описатели нумеруются натуральными числами. Первый описатель закреплён за «файлом» плохих блоков. Второй описывает корневой каталог ФС. Назначение прочих описателей не имеет фиксированного предназначения. Зная номер и размер описателя нетрудно вычислить его координаты на диске.

Структура индексного дескриптора изображена на рисунке:



1. Тип и права доступа.
2. Число ссылок (счётчик числа ссылок на файл).
3. Идентификатор владельца.
4. Идентификатор группы, к которой принадлежит владелец.
5. Размер файла в байтах.
6. Время последнего доступа.
7. Время последней записи.
8. Время последней модификации inode.
9. Размер файла в блоках.

Структура адресной информации inode.



10 – первого уровня
11 – второго уровня
12 – третьего уровня
На указатель отводится 4 байта.
(80 кб. + 2048)*8192 – максимальный размер файла.

Если блок 8 кб, то максимальная длина файла, использующего прямую адресацию, равна 80 кб.

Косвенная адресация первого уровня ссылается на блок. На указатель отводится 4 байта. Всего в блоке может быть 2048 ссылок. Косвенная адресация второго уровня: блок содержит 2048 ссылок на блоки, имеющие ссылки.

Недостатки:

1. «Дальние перемещения» (суперблок далеко от последнего файла в системе) → концепция цилиндров (ufs).
2. Внутренняя фрагментация (каждому файлу выделяется целое число блоков; последний блок используется не полностью) → блоки и фрагменты (ufs).
3. «Маленькие файлы» → проблема не решена в S5, ufs.

Концепция цилиндров.

После создания файловой системы файлы записываются в последние блоки. В дальнейшем (когда файлы создаются, удаляются, изменяются) файлы занимают любые свободные блоки. Таким образом, файл может быть разбросан по всему диску. Таблица индексных дескрипторов находится в начале ФС, поэтому по мере заполнения ФС и увеличения фрагментации всё больше времени тратится на частые и дальние перемещения головок чтения/записи. Для устранения этого недостатка используют концепцию групп цилиндров.

Первоначально эта концепция появилась в ФС ufs. По умолчанию ufs использует группы цилиндров, состоящие из 16 цилиндров. Каждая группа цилиндров описывается своим блоком группы цилиндров.

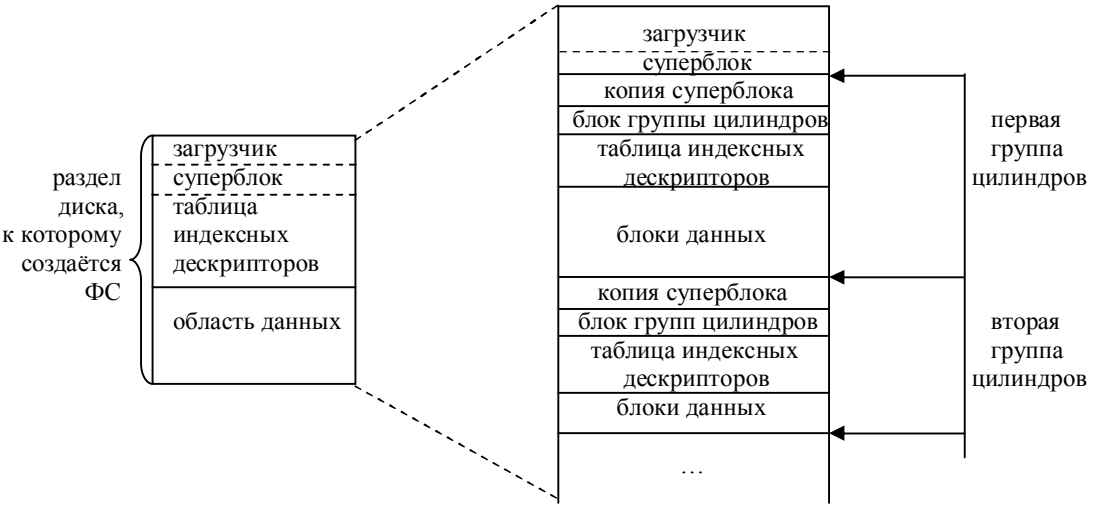
Файлы по-прежнему остаются фрагментированными, однако схема, которую использует ФС ufs, значительно сокращает фрагментацию по сравнению с S5 (где данная концепция не используется).

ФС стремится размещать каталоги и входящие в них файлы в одной и той же группе цилиндров. Таким образом, чтобы, например, прочитать файл, потребуется переместить головки максимум на 16 цилиндров. Большие файлы распределяются между группами цилиндров так, чтобы занимать не более 2 мегабайт в каждой из групп. Это предотвращает заполнение группы цилиндров одним файлом. Выигрыш достигается за счёт того, что дальние перемещения головок осуществляются только после того, как прочиталось или записалось 2 Мб информации.

Эффективность схемы размещения файлов падает, если системе не хватает места для перемещения информации. Процессы чтения и записи замедляются, если свободно менее 10% ФС.

Необходимый запас автоматически резервируется ОС и только суперпользователь имеет право его использовать.

Иллюстрация к концепции цилиндров:



Блоки и фрагменты (ufs).

Преимуществом большого блока является то, что ускоряется обмен данными с диском при передаче больших объёмов информации.

Недостаток: файлы больших размеров неэкономно используют дисковое пространство.

Для борьбы с внутренней фрагментацией используется метод разбиения блока на фрагменты, которые можно распределять таким образом, чтобы файл мог и не занимать весь блок целиком.

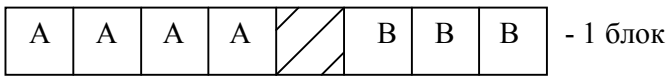
Размер фрагмента не меньше размера сектора.

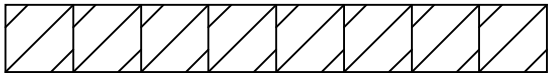
Sun OC:

ufs → блок 8192 байта

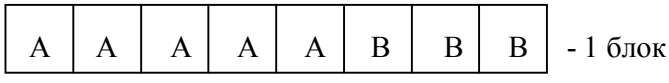
ufs → фрагмент 1024 байта

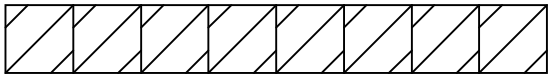
Пример:



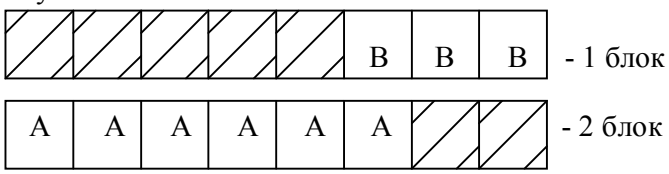


A увеличился на 1 кб

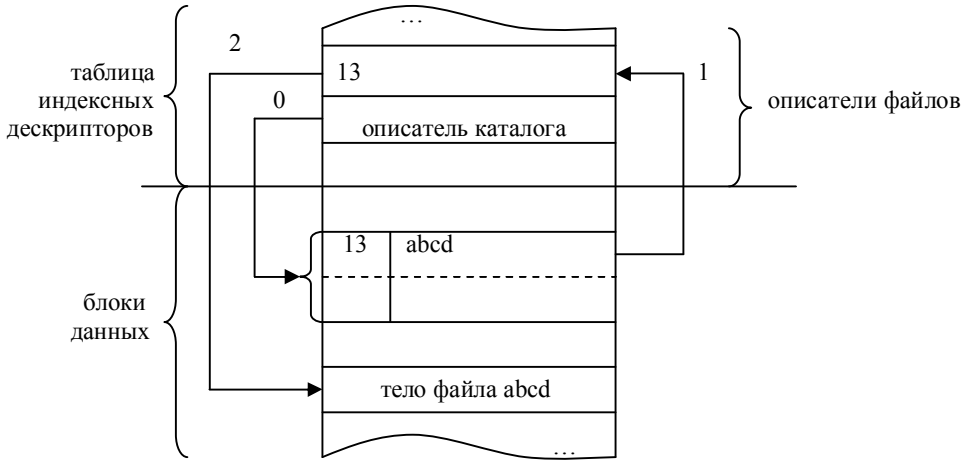




А увеличился на 1 кб



Взаимосвязь между элементами каталогов и описателями файлов.



элемент каталога:

номер описателя	имя файла

Другой информации о файле в элементе каталога нет.

На один описатель могут существовать ссылки из нескольких элементов одного или нескольких каталогов одной ФС. С точки зрения ФС любой каталог представляет собой обычный файл со своим описателем. Содержимое каталога располагается в области блоков данных.

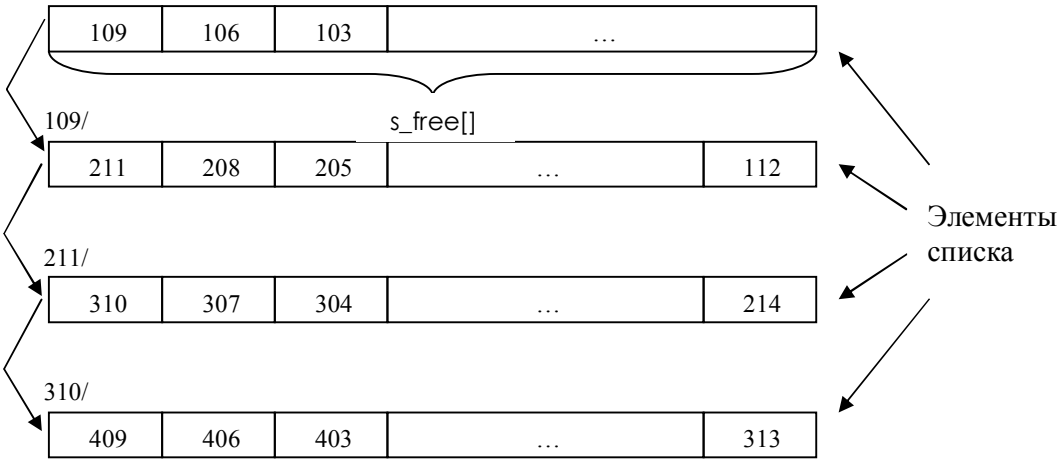
Выделение дисковых блоков.

Каждый раз, когда данные записываются в файл, ядро должно выделить из файловой системы дисковые блоки под информационные блоки прямой либо косвенной адресации, связав их с описателем файлов в таблице индексных дескрипторов.

Суперблок содержит массив s_free[], в котором содержатся номера свободных дисковых блоков. Программа ядра **mkfs** организует хранение неиспользуемых в данный момент дисковых блоков в виде списковой структуры, первым элементом которой является массив s_free[]. Каждый элемент этой списковой структуры хранит номера свободных дисковых блоков аналогично массиву s_free[], при этом самый первый номер в каждом элементе является ссылкой на следующий элемент списковой структуры.

Изначально программа **mkfs** заполняет массив s_free[] и другие элементы списковой структуры таким образом, чтобы номера дисковых блоков были упорядочены.

Впоследствии, в связи с непредсказуемостью запросов и освобождений дисковых блоков, упорядоченность нарушается.



Первый номер массива (блоки №№109, 211, 310 и 409) – указатель на следующий элемент списка.

Когда ядру необходимо выделить блок, выполняется алгоритм **alloc**, который выделяет очередной дисковый блок, номер которого содержится в массиве s_free[].

Если выделенный блок является последним в массиве s_free[], ядро трактует его как указатель на некоторый дисковый блок, в котором содержатся номера дисковых блоков.

Ядро переписывает содержимое этого блока в массив s_free[] суперблока, после чего выделяет запрошенный блок для целей файловой системы.

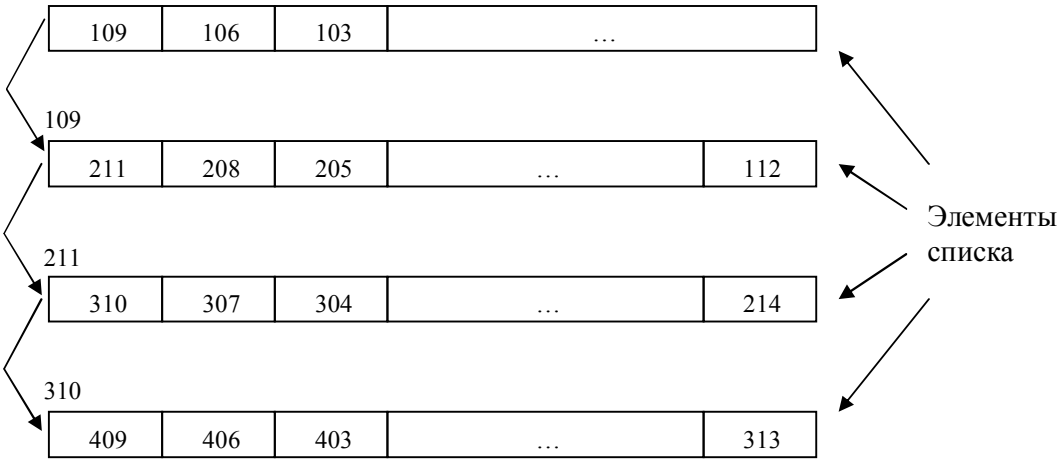
Алгоритм alloc:

1. Если суперблок заблокирован, приостановиться до тех пор, пока не будет снята блокировка.
2. Удалить блок из списка свободных блоков суперблока.
3. Если из списка удалён последний блок:
   - заблокировать суперблок;
   - прочитать блок, только что взятый из списка свободных блоков;
   - скопировать номера блоков, хранимых в этом блоке, в массив s_free[] суперблока;
   - снять блокировку суперблока.
4. Переписать в буфер блок, удалённый из массива s_free[].
5. Уменьшить общее число свободных блоков.
6. Пометить суперблок как «изменённый».
7. Возвратить буфер, содержащий выделенный блок.

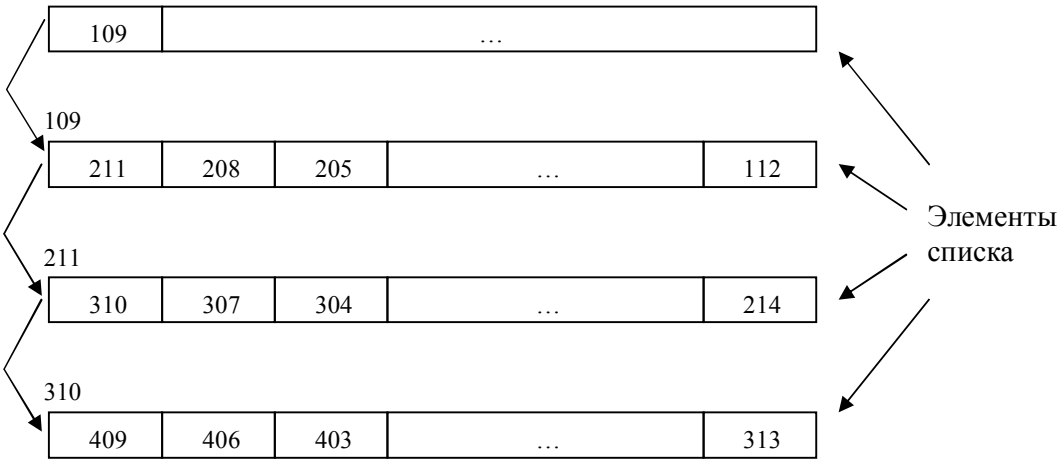
При выполнении алгоритма освобождения блока free, если список в массиве s_free[] суперблока не полон, номер освобождённого блока добавляется в массив s_free[]; если список полон, освобождаемый блок становится связанным блоком в списке, ядро переписывает в этот блок содержимое массива s_free[] и номер освобождённого блока становится единственным элементом массива s_free[].

Пример манипуляции с выделением и освобождением дисковой памяти:

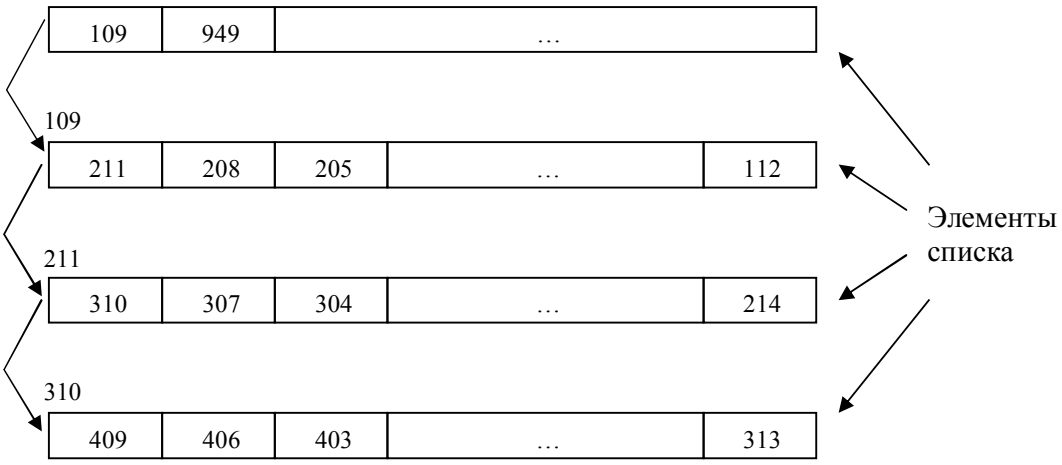
а) начальное состояние:



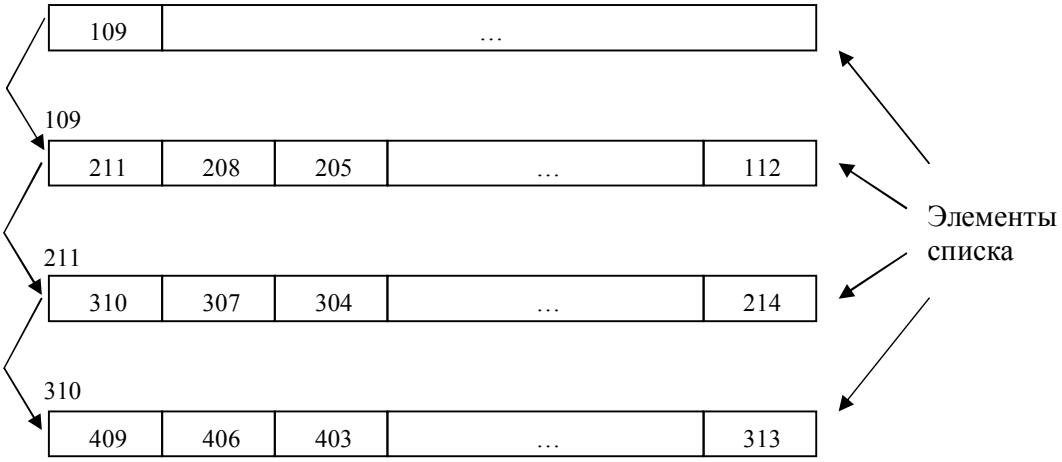
б) после запроса двух блоков:



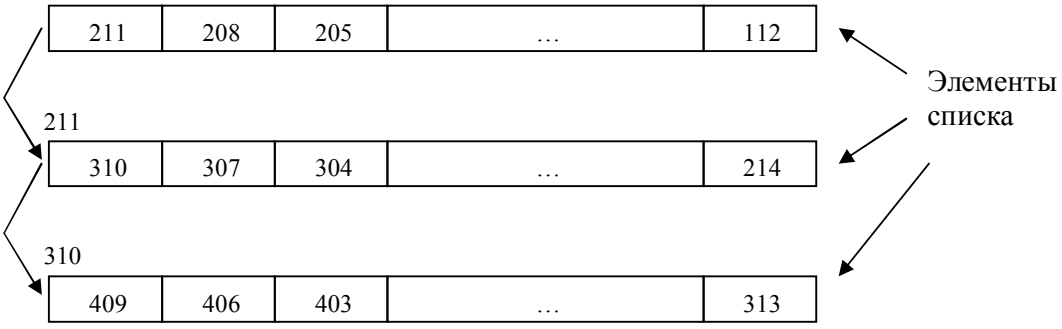
в) после освобождения блока с некоторым номером (пусть 949):



г) после запроса очередного блока:



д) запрос ещё одного блока:



е) освобождение некоторого блока (пусть 504):

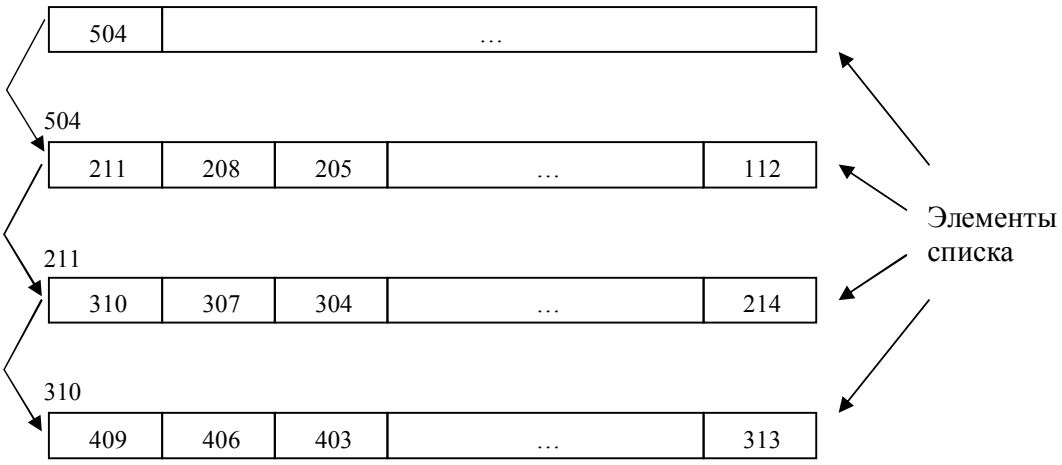


Таблица описателей файлов содержит линейный список индексных дескрипторов, при этом описатель файла считается свободным, если поле его типа содержит нулевое значение. Массив s_inode[] в суперблоке содержит информацию о свободных описателях файлов и используется аналогично массиву s_free[]. Разница между s_free[] и s_inode[]: поскольку положение описателя файлов известно, и описатель файла сам содержит в себе информацию, свободен он или нет, то списковая структура не используется.

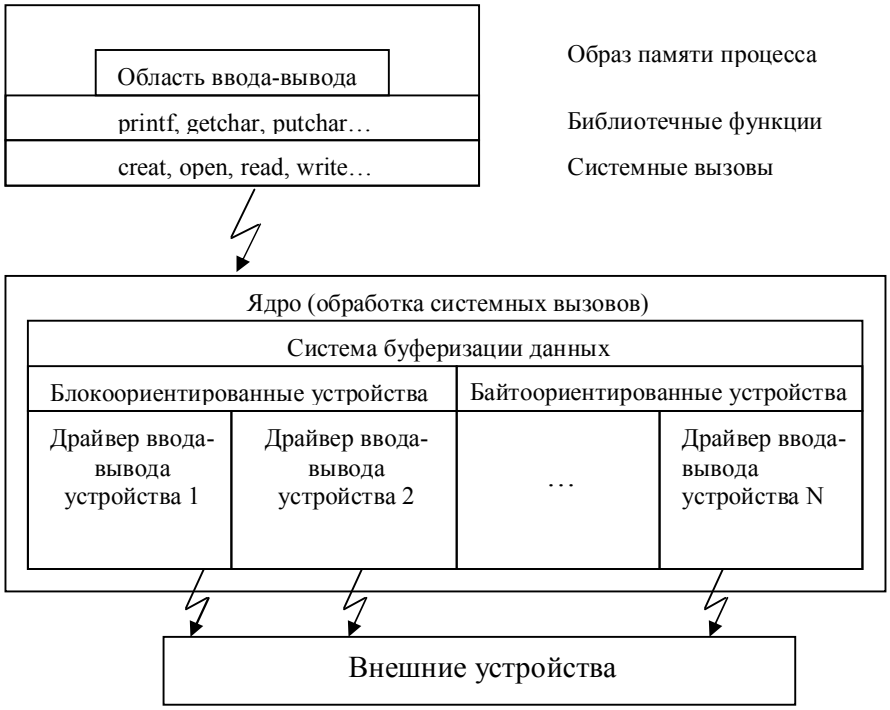
Если массив s_inode[] в суперблоке не пуст, ядро выделяет очередной описатель файла и назначает его некоторому файлу. Если массив пуст, ядро просматривает таблицу описателей файлов, выбирая из неё номера свободных описателей файлов, и заполняет ими массив s_inode[], причём запоминается та точка таблицы описателей файлов, на которой закончен просмотр с тем, чтобы при следующем заполнении массива s_inode[] начать с этой точки.

Системные операции для работы с файловой системой.

Открытие файла – процесс установления связи между именем файла и некоторой системной переменной, хранимой в памяти процесса, называемой идентификатором файла.

Каждый раз, когда интерпретатор запускает программу, он открывает три файла с идентификаторами 0, 1, 2 (stdin – стандартный ввод, stdout – стандартный вывод, stderr – стандартный поток ошибок).

Система ввода-вывода представима в виде нескольких уровней:



Работа этой системы поддерживается тремя основными структурами (таблицами):

- таблица описателей файлов;
- таблица файлов;
- таблица открытых файлов (таблица пользовательских дескрипторов файлов).

Таблица описателей файлов – структура, элементами которой являются копии описателей файлов: по одной на каждый файл, к которому осуществлена попытка доступа.

При выполнении операции открытия файлов по полному имени файла определяется элемент каталога, по нему определяется номер описателя файла, и этот описатель файла копируется в данную структуру (таблицу описателей файлов).

Все последовательные изменения файла фиксируются в элементе таблицы файлов до тех пор, пока файл не будет закрыт. При закрытии файла изменённый элемент таблицы описателей файлов копируется обратно в таблицу индексных дескрипторов.

При каждом открытии файла создаётся новый элемент в таблице файлов, при этом одному элементу таблицы описателей файлов соответствует один или несколько элементов в таблице файлов. Каждый элемент в таблице файлов содержит информацию о режиме открытия файла, о положении указателя чтения-записи и о числе ссылок на данный элемент.

Число ссылок увеличивается при наследовании (получении копии) дескриптора файла и при создании нового процесса, который наследует контекст среды своего предка.

Таблица открытых файлов.

Элемент таблицы открытых файлов содержит номер дескриптора файла и ссылку на элемент таблицы файлов.

При создании нового процесса, создаётся копия элементов таблицы открытых файлов родителя.

Основные операции.

1. Системный вызов open().

```
#include <sys/types.h>
#include <fcntl.h>

fd = open (char *pathname, int flag[, mode_t mode]);
```

fd – дескриптор файла или -1 (ошибка);
pathname – имя открываемого или создаваемого файла;
flag – режим;
mode – права для создаваемого файла.

Режимы:
O_RDONLY (0) – открытие файла только для чтения;
O_WRONLY (1) – открытие файла только для записи;
O_RDWR (2) – открытие файла для чтения и для записи;
O_APPEND – добавление информации в конец файла;
O_CREAT – создание файла;
O_TRUNC – уменьшение длины файла до 0.

Права представляют собой трёхзначное восьмеричное число, в котором первая цифра соответствует владельцу файла, вторая – группе, к которой принадлежит владелец, третья – всем прочим. Каждая из цифр является комбинацией масок: 4 – чтение, 2 – запись, 1 – выполнение.

Пример: 0644 – владелец – RW (чтение и запись), группа и прочие – R (только чтение).

Режимы могут задаваться одновременно через операцию ‘|’.

Не стоит путать системный вызов **open()** и библиотечную функцию **fopen()** с режимами только для записи; **open()** по умолчанию не уменьшает размер файла до 0. Если мы запишем 10 байт в файл из 500 байт, то в файле изменятся только первые 10 байт, остальные останутся неизменными. Библиотечная функция **fopen()** по умолчанию уменьшает размер файла до 0. Проделав описанные выше действия, мы получим файл, содержащий *только* записанные 10 байт.

Алгоритм open():

1. Превратить имя файла в описатель файла.
2. Если файл не существует, и при этом отсутствует третий аргумент, либо не разрешён доступ, возвратить -1 (код ошибки).
3. Выделить для описателя файла запись в таблице описателей файлов, инициализировать счётчик и смещение.
4. Выделить запись в таблице файлов в соответствии с режимом открытия файла, инициализировать счётчик и смещение, установить указатель на запись в таблице описателей файлов.
5. Выделить запись в таблице открытых файлов процесса. Инициализировать счётчик и смещение, установить указатель на запись в таблице файлов.
6. Возвратить пользовательский дескриптор файла.

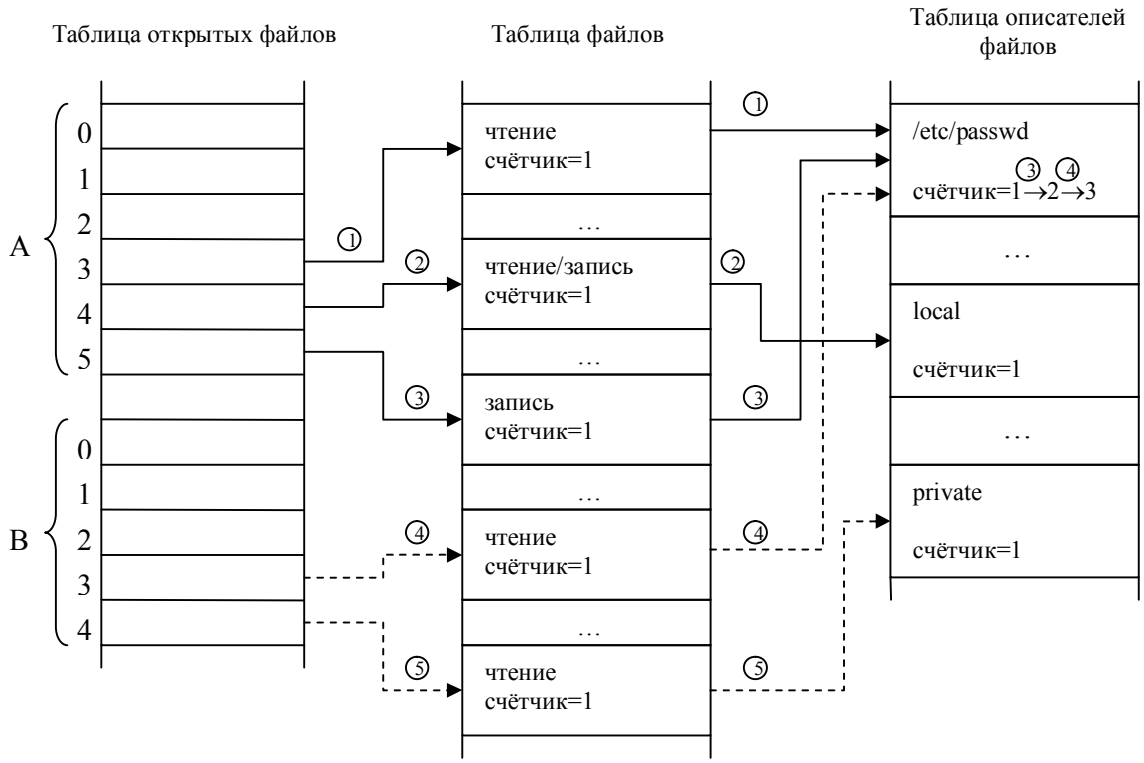
Пример:

Процесс А открывает файл /etc/passwd дважды: для чтения и для записи, и файл local для чтения и записи.

fd1 = open (“/etc/passwd”, O_RDONLY); fd1=3
fd2 = open (“local”, O_RDWR); fd2=4
fd3 = open (“/etc/passwd”, O_WRONLY); fd3=5

Пусть процесс В открывает ещё два файла:

fd1 = open (“/etc/passwd”, O_RDONLY); fd1=3
fd2 = open (“private”, O_RDONLY); fd2=4



2. Системный вызов creat() – создание файла с определённым именем и правами.

```
#include <sys/types.h>
#include <fcntl.h>
fd = creat (char *pathname, mode_t mode);
```

Параметры функции аналогичны соответствующим параметрам системного вызова open().

Пример:

```
fd = creat (“/tmp/newfile”, 0644);
```

или то же самое

```
fd = open (“/tmp/newfile”, O_CREAT | O_WRONLY, 0644);
```

Алгоритм creat():

1. Получить описатели файлов по данному имени файла (для указанного имени файла).
2. Если файл существует, и доступ к нему не разрешён, освободить описатель файла и возвратить код ошибки. Иначе считаем, что файл не существует:
   - назначить свободный описатель файла;
   - создать новую точку входа в родительский каталог;
   - включить имя нового файла в оглавление.
3. Выделить для описателя файла запись в таблице описателей файлов и инициализировать счётчик и смещение.
4. Выделить запись в таблице файлов в соответствии с режимом открытия, инициализировать счётчик и смещение, установить указатель на запись в таблице описателей файлов.
5. Выделить запись в таблице открытых файлов процесса, инициализировать счётчик и смещение, установить указатель на запись в таблице файлов.
6. Возвратить пользователю дескриптор файла.

3. Системные вызовы read() и write() – чтение и запись из/в файл.

```
#include <unistd.h>

ssize_t n_read = read (int fd, void *buf, size_t n);
ssize_t n_write = write (int fd, void *buf, size_t n);
```

n_read, n_write – фактическое число переданных байт;
fd – дескриптор файла;
buf – буфер, откуда/куда выполняется операция чтения/записи (запись void * говорит о том, что тип передаваемых данных может быть любым);
n – предполагаемое число передаваемых байт.

Если n_read не равно n, то, вероятно, что произошла ошибка.
ssize_t – целое, int.

Текущее положение очередного читаемого/записываемого байта отслеживается посредством объекта, носящего название *указателя файла* (указатель чтения/записи, pointer read/write, pointer file), который хранится в элементе таблицы файлов. Значение этого указателя определяется режимом открытия файла.

Пример:
Записываем 10 байт в существующий файл (500 байт), открытый различным образом.

```
filedes = open ("oldfile", O_WRONLY);
```

Заменяются только первые 10 байт, размер файла по-прежнему 500 байт.

```
filedes = open ("oldfile", O_WRONLY | O_APPEND);
```

10 байт дописываются в конец файла, размер файла 510 байт.

Пример:

Реализация getchar() средствами нижнего уровня.

```
#include <unistd.h>
#define BUFSIZE 512
int getchar() /*вариант с буфером*/
{
    static char buf[BUFSIZE];
    static char *bufp = buf;
    static int n = 0;
    if (n == 0) /*буфер пуст*/
    {
            n = read (0, buf, BUFSIZE);
            bufp = buf;
    }
    return ((n--) > 0)? *bufp++ : EOF);
}
```

Процесс может открыть файл более 1 раза и читать из него данные, используя разные файловые дескрипторы (разные записи в таблице файлов).

Ядро работает с указателями чтения/записи, хранящимися в разных элементах таблицы файлов, независимо друг от друга.

Пример:

```
#include <fcntl.h>
void main()
{
    int fd1, fd2;
    char buf1[512], buf2[512];
    fd1 = open ("/etc/passwd", O_RDONLY);
    fd2 = open ("/etc/passwd", O_RDONLY);
    read (fd1, buf1, sizeof(buf1));
    read (fd2, buf2, sizeof(buf2));
}
```

После выполнения данного кода buf1=buf2.

Алгоритм read():

1. Обратиться к записи таблицы файлов по значению дескриптора файла.
2. Проверить доступность файла.
3. Установить параметры в адресном пространстве процесса.
4. Получить запись таблицы описателей файлов по записи в таблице файлов.
5. Заблокировать запись в таблице описателей файлов.
6. Установить значение смещения в байтах для адресного пространства процесса по значению смещения в таблице файлов.
7. Пока значение счётчика байтов не станет удовлетворительным:
   - превратить значение смещения в файле в номер дискового блока;
   - вычислить смещение внутри блока и количество байт, которые необходимо прочитать;
   - если количество байт при чтении равно нулю, прерваться и выйти из цикла;
   - прочитать блок;
   - скопировать данные из системного буфера по адресу пользователя;
   - скорректировать значения полей в адресном пространстве;
   - освободить буфер.
8. Разблокировать запись в таблице описателей файлов.
9. Скорректировать значение смещения в таблице файлов для последующей операции чтения.
10. Возвратить общее число прочитанных байт.

4. Системный вызов close().

close (fd); - закрытие файла.

Если счётчик ссылок в таблице файлов имеет значение больше единицы, то это означает, что на запись в таблице файлов имеют ссылки другие пользовательские дескрипторы, ядро уменьшает значение счётчика на единицу. Если значение счётчика равно 1, запись в таблице файлов удаляется.

При удалении записи в таблице файлов уменьшается число ссылок на запись в таблице описателей файлов. Если на запись в таблице описателей файлов ссылалась всего одна запись в таблице файлов, запись в таблице описателей файлов удаляется.

В любом из этих случаев удаляется запись в таблице открытых файлов.

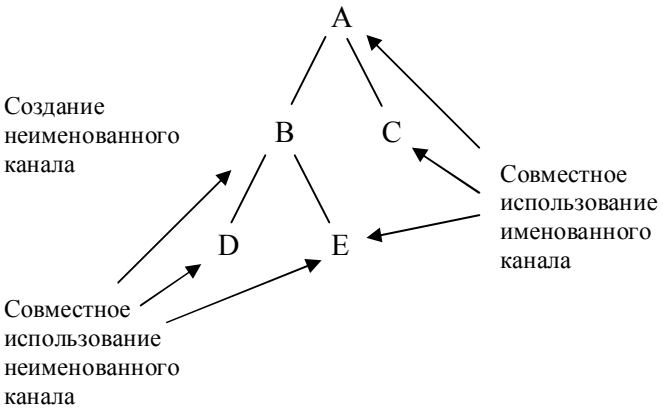
Каналы.

Существуют каналы двух видов: неименованные и именованные.

Неименованные каналы являются временными, создаются системным вызовом pipe() и могут использоваться только потомками того процесса, который их создал.

Именованные каналы с точки зрения файловой системы являются обычными файлами, которые открываются системным вызовом open(), и могут использоваться любым процессом.

С точки зрения системных вызовов read/write эти виды каналов равнозначны.



Неименованный канал:

Системный вызов pipe() возвращает два дескриптора – для чтения и для записи.

```
#include <unistd.h>
int pipe (int filedes[2]);
```

filedes[0] – дескриптор чтения из канала;

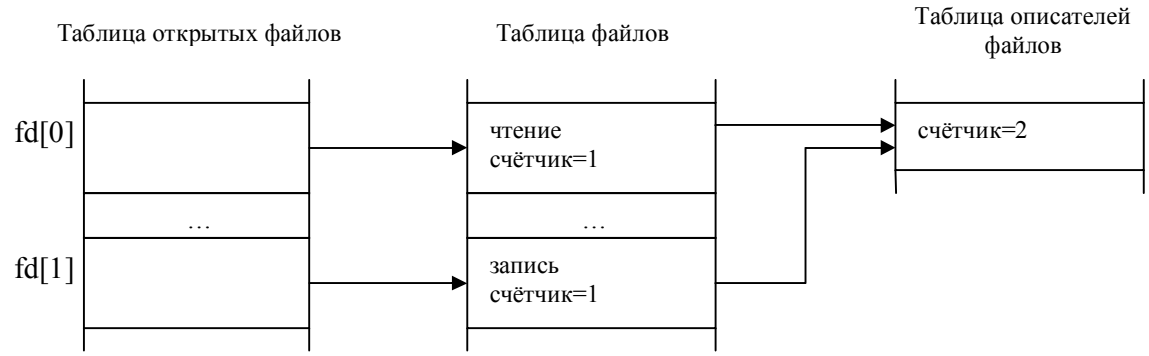
filedes[1] – дескриптор записи в канал;

Чтение и запись осуществляются обычными системными вызовами read() и write(). Система может приостановить процесс, выдавший один из этих системных вызовов, если чтение производится из пустого канала или запись производится в полный канал.

Алгоритм pipe():

1. Назначить новый описатель файла для канала.
2. Выделить одну запись в таблице файлов для чтения и одну для записи.
3. Инициализировать записи в таблице файлов таким образом, чтобы они указывали на новый созданный описатель файла.
4. Выделить один пользовательский дескриптор файлов для чтения и один для записи, инициализировав их таким образом, чтобы они указывали на соответствующие записи в таблице файлов.

5. Установить значение счётчика в записи таблицы описателей файлов, равное двум, а значение счётчиков в таблице файлов, равное единице.



Пример:
Чтение/запись в канал:

```
char string[] = "Example1 with a pipe";

int main()
{
    char buf[1024];
    char *cp1, *cp2;
    int fds[2];
    cp1 = string;
    cp2 = buf;
    while (*cp1)
            *cp2++ = *cp1++;
    pipe (fds);
    for (;;)
    {
            write (fd[1], buf, 20);
            read (fd[0], buf, 20);
    }
}
```

Именованный канал:

```
#include <sys/types.h>
#include <sys/stat.h>
```

Создание именованного канала можно произвести двумя системными вызовами:

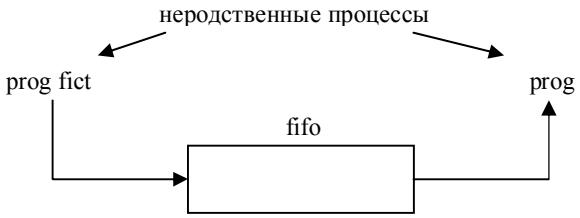
```
int mkfifo (char* pathname, mode_t mode);
int mknod (char* pathname, mode_t mode, int dev);
```

pathname – имя файла
mode – права доступа (последние три байта) и специальные режимы:
010000 – создание именованного канала
020000 – создание специального байтоориентированного файла
040000 – создание каталога
060000 – создание специального блокоориентированного файла
dev – при создании именованного канала должно принимать нулевое значение.
По умолчанию права доступа 0666.

Пример чтения/записи в именованный канал:

```
#include <fcntl.h>

char string[] = "Example2 with a fifo";

int main (int argc, char* argv[])
{
    int fd;
    char buf[1024];
    mknod ("fifo", 010666, 0);
    if (argc == 2)
            fd = open ("fifo", O_WRONLY);
    else
            fd = open ("fifo", O_RDONLY);
    for (;;)
    {
            if (argc == 2)
                    write (fd, string, 20);
            else
                    read (fd, buf, 20);
    }
}
```

Программа, запущенная с каким-либо параметром, будет бесконечно записывать в канал строку string, а программа, запущенная без параметра, будет бесконечно читать из канала 20 символов.



fict – параметр, с которым запущен процесс prog.

Наследование пользовательских дескрипторов.

Системный вызов **dup()** создаёт копию указанного дескриптора файла, возвращая дескриптор с наименьшим номером из имеющихся в системе.

```
#include <fcntl.h>
int newfd = dup (fd);
```

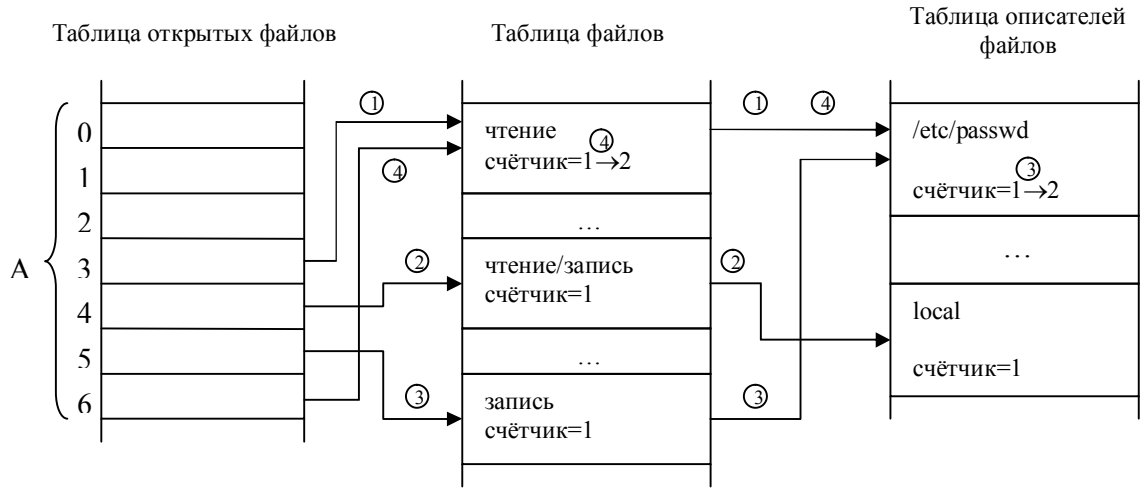
fd – номер существующего дескриптора

Старый и новый дескрипторы в таблице открытых файлов ссылаются на одну и ту же запись в таблице файлов, следовательно, используют одно и то же смещение для чтения или записи в файл.

Пример:

```
fd1 = open ("/etc/passwd", O_RDONLY);      (1)   fd1=3
fd2 = open ("local", O_RDWR);              (2)   fd2=4
fd3 = open ("/etc/passwd", O_WRONLY);      (3)   fd3=5
fd4 = dup (fd1);                           (4)   fd4=6
```

(такие номера получаем только при условии, что ранее не было открыто других файлов или не было закрыто каких-либо стандартных файлов)



После выполнения данных действий мы можем читать файл **/etc/passwd** как при помощи fd1, так и при помощи fd4.

Пример (чтение из файла /etc/passwd при помощи двух дескрипторов файла):

```
int main()
{
    int i, j;
    char buf1[512], buf2[512];
    i = open ("/etc/passwd", O_RDONLY);
    j = dup (i);
    read (i, buf1, sizeof (buf1)); //Читаем из файла 512 байт
    read (j, buf2, sizeof (buf2)); //Читаем следующие 512 байт
    close (i);
    read (j, buf2, sizeof (buf2)); //Продолжаем считывать информацию из того же файла
}
```

Альтернативный способ копирования дескрипторов:

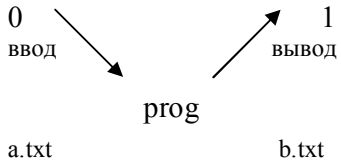
```
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

int fcntl (int fd, int command, int argument);
```

Функция fcntl() выполняет действия по разделению пользовательских дескрипторов в соответствии со значением аргумента command, который определён в файле fcntl.h. При значении второго аргумента, равного F_DUPFD системный вызов **fcntl()** возвратит первый свободный дескриптор, значение которого не меньше третьего аргумента, и который является копией первого аргумента.

Основным назначением системных вызовов dup() и fcntl() является переназначение ввода/вывода для каналов и стандартных файлов ввода/вывода.

Пример:

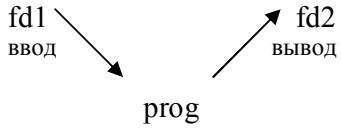
Некоторая программа читает данные из стандартного ввода и выводит результаты на стандартный вывод. Необходимо, чтобы эта программа читала данные из файла a.txt и выводила в файл b.txt.




```
#include <fcntl.h>
……..
int fd1, fd2;
fd1 = open ("a.txt", O_RDONLY);
close (0);
fcntl (fd1, F_DUPFD, 0);
fd2 = open ("b.txt", O_WRONLY);
close (1);
fcntl (fd2, F_DUPFD, 1);
```

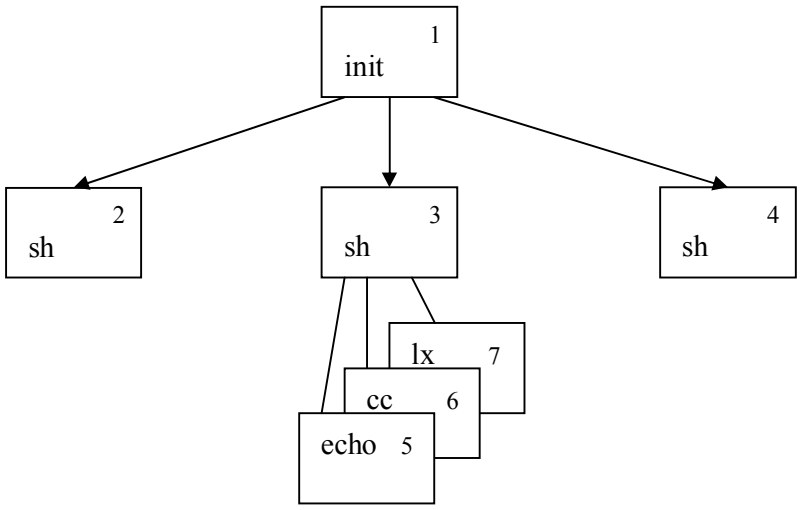
0 ← fd1

1 ← fd2



Перенаправление стандартного ввода/вывода.

Управление процессами.



Порождение нового процесса осуществляется с помощью системного вызова fork().

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork();
```

Системный вызов **fork()** – единственное средство порождения процессов.

Порождённый процесс является почти полной копией исходного процесса и отличается от него только номером (идентификатором) процесса. Созданный процесс наследует от родительского процесса контекст среды, включая дескрипторы файлов, программные каналы и т.д. Системный вызов fork() возвращает в исходный процесс идентификатор порождённого процесса, а в порождённый процесс – 0. В случае неуспешной попытки порождения процесса в родительский процесс возвращается -1.

Процесс может получить свой идентификатор посредством системного вызова **getpid()**, а при помощи системного вызова **getppid()** – идентификатор родительского процесса:

```
pid = getpid();
pid = getppid();
```

Существует средство объединения процессов в группы. После порождения процесса, дочерний процесс наследует в том числе и код группы своего родителя. После этого порождённый процесс может создать новую группу процессов, либо присоединиться к существующей группе. Для этого существует два альтернативных вызова:

```
int setpgip (pid_t pid, pid_t pgip);
int setpgrp (pid_t pid, pid_t pgip);
```

Данные системные вызовы абсолютно одинаковы и существуют в целях совместимости.

Системный вызов устанавливает идентификатор группы процессов (второй аргумент) на основании идентификатора первого аргумента. Если pid равно нулю, идентификатор группы процессов задаётся равным идентификатору выдавшего данный системный вызов процесса.

Если оба аргумента одинаковы, то процесс становится лидером группы.

При помощи системного вызова **getpgip()** можно получить идентификатор группы процессов:

```
pid_t getpgip();
```

После создания процесса-потомка, его приоритет равен приоритету родительского процесса. Изменить этот приоритет можно системным вызовом **nice()**:

```
int nice (int number);
```

Если number больше нуля, то приоритет уменьшается на заданную величину, если же number меньше нуля, то приоритет увеличивается на заданную величину, однако увеличивать приоритет можно только в режиме суперпользователя (root).

Алгоритм fork():

1. Проверить доступность ресурсов ядра.
2. Получить свободное место в таблице процессов и уникальный идентификатор процесса (PID).
3. Проверить, не запустил ли пользователь слишком много процессов (не превышено ли ограничение).
4. Сделать пометку, что порождённый процесс находится в состоянии создания.
5. Скопировать информацию в таблицу процессов из записи, соответствующей родительскому процессу, в запись, соответствующую порождаемому процессу.
6. Увеличить значение счётчика открытых файлов в таблице файлов.
7. Сделать копию контекста родительского процесса
8. Если в данный момент выполняется родительский процесс, то:
   - перевести порождённый процесс в состояние готовности;
   - возвратить идентификатор процесса в родительский процесс.
9. Если выполняется порождённый процесс, то:
   - записать начальные значения в поля синхронизации адресного пространства.

Замечания:

1. Количество процессов, одновременно запущенных одним пользователем, ограничивается на уровне конфигурации ОС. Кроме этого, обычным пользователям не разрешается создавать процесс, занимающий последнюю запись в таблице процессов.
2. Порождённый процесс не только наследует права доступа к открытым файлам, но и разделяет доступ к этим файлам с родительским процессом.
3. Поскольку дескрипторы файлов исходного и порождённого процессов ссылаются на одну и ту же запись в таблице файлов, оба процесса никогда не осуществят доступ к одной и той же области памяти, поскольку работают с одним и тем же указателем чтения/записи.

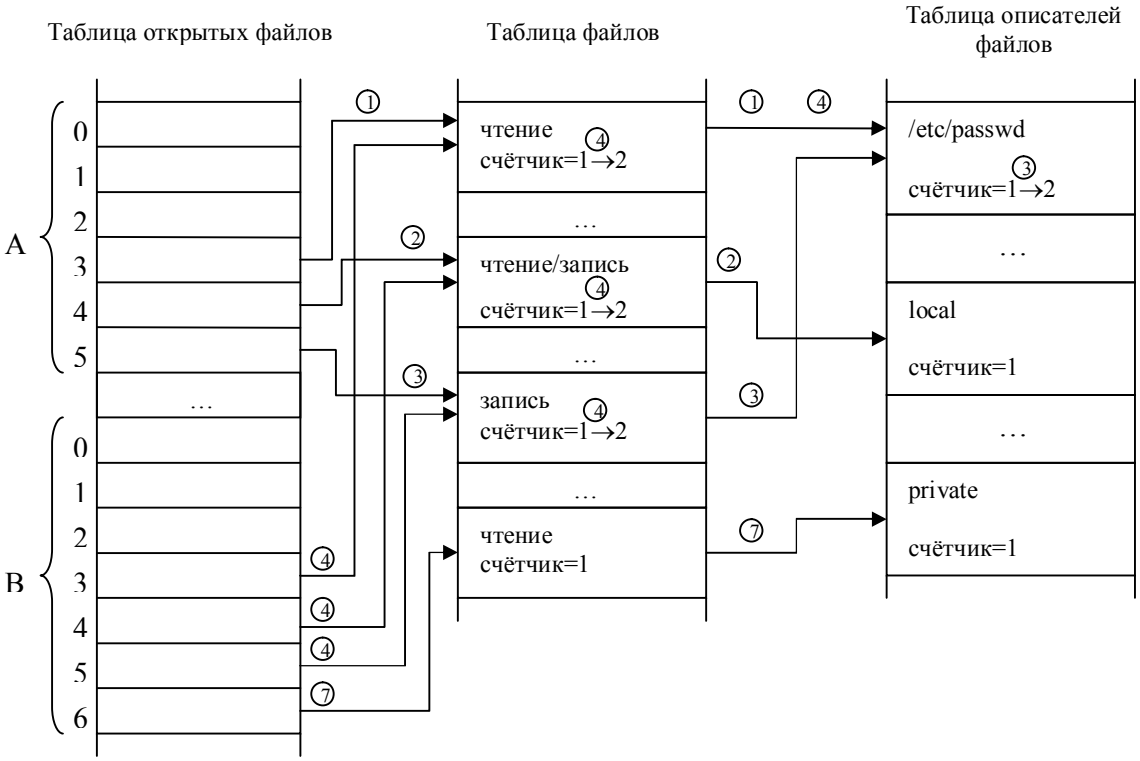
Пример:

Процесс A.

```
fd1 = open ("/etc/passwd", O_RDONLY);        (1)    fd1=3
fd2 = open ("local", O_RDWR);                (2)    fd2=4
fd3 = open ("/etc/passwd", O_WRONLY);        (3)    fd3=5
fork();                      (4)          создаём процесс B
```

Процесс B.

```
fd4 = open ("private", O_RDONLY);      (5)          fd4=6
```



Завершается процесс системным вызовом **exit()**:

void exit (int status);

Единственный аргумент – так называемый статус завершения процесса, младшие 8 бит которого могут быть доступны родительскому процессу, при условии, что родительский процесс выполнил системный вызов **wait()**.

Процесс может выполнить exit() как в явном виде, так и в неявном (т.е. после завершения процедуры main() автоматически вызывается системный вызов exit()).

Ядро может вызвать exit() по собственной инициативе, если процесс не принял посланный ему сигнал (в таком случае статус завершения процесса будет равен номеру посланного, но не принятого сигнала).

Алгоритм exit():

1. Игнорировать все поступающие процессу сигналы.
2. Закрыть все открытые файлы.
3. Освободить области и память, ассоциированные с процессом.
4. Создать запись учётной информации.
5. Прекратить существование процесса.
6. Назначить всем процессам-потомкам в качестве родителя процесс init().
7. Если какой-либо из потомков прекратил существование, то:
   послать процессу init сигнал гибели потомка.
8. Переключить контекст.

Пример 1:

Программа, в которой родительский и порождённый процессы разделяют доступ к файлу.

```
#include <fcntl.h>

int fdrd, fdwt;
char c;

void main (int argc, char* argv[])
{
    if (argc != 3)
            exit (1);
    if ((fdrd = open (argv [1], O_RDONLY)) == 1)
            exit (1);
    if ((fdwd = creat (argv[2], 0666)) == 1)
            exit (1);
    fork();
    /*Оба процесса используют одну и ту же программу*/
    for (;;)
    {
            if (read (fdrd, &c, 1) != 1)
                    break;
            write (fdwt, &c, 1);
    }
    exit (0);
}
```

Результатом может быть как копия исходного файла, так и файл, составленный из почти случайно перемешанных символов исходного файла.

Пример 2:

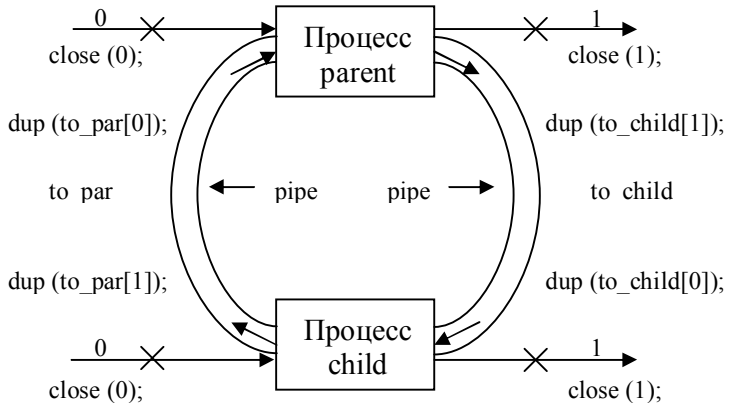
Программа, в которой родительский и порождённый процессы не разделяют доступ к файлам.

Примером может служить программа, представленная выше, в которой fork() перенесён на две строки вверх.

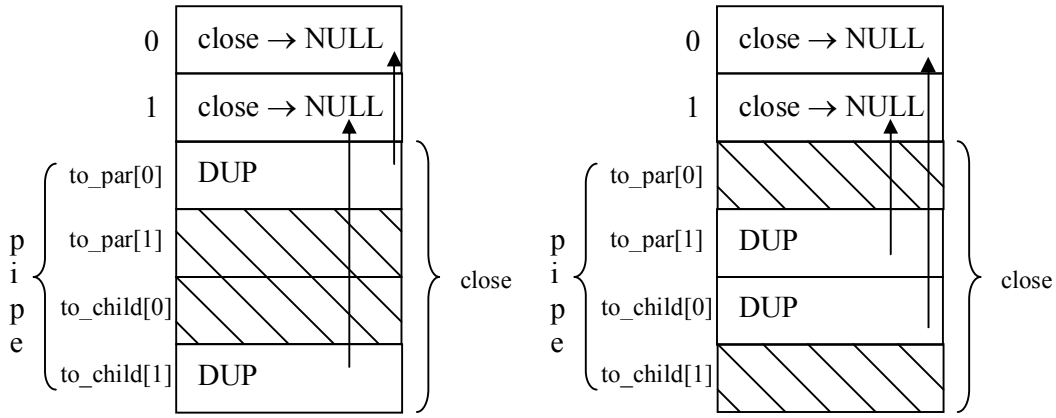
Пример 3:

Разделение процессами канала.

```
#include <string.h>

char string[] = "Разделение канала";

void main ()
{
    int count, i;
    int to_par[2], to_child[2]; /*Для каналов родителя и потомка*/
    char buf[256];
    pipe (to_par);
    pipe (to_child);
    if (fork() == 0)
    {
/*Выполнение порождённого процесса*/
        close (0); /*Закрытие стандартного ввода*/
        dup (to_child[0]); /*Дублирование дескриптора чтения из канала в позицию стандартного ввода*/
        close (1); /*Закрытие стандартного вывода*/
        dup (to_par[1]); /*Дублирование дескриптора записи в канал в позицию стандартного вывода*/
/*Закрытие ненужных дескрипторов канала*/
        close (to_par[0]);
        close (to_par[1]);
        close (to_child[0]);
        close (to_child[1]);
        for (;;)
        {
            if ((count = read (0, buf, sizeof (buf))) == 0)
                exit ();
            write (1, buf, count);
        }
    }
/*Выполнение родительского процесса*/
/*Перестройка стандартного ввода/вывода*/
    close (1);
    dup (to_child[1]);
    close (0);
    dup (to_par[0]);
    close (to_par[0]);
    close (to_par[1]);
    close (to_child[0]);
    close (to_child[1]);
    for (i=0, i<15; i++)
    {
        write (1, string, strlen (string));
        read (0, string, strlen (string));
    }
}
```



Таблицы открытых файлов процессов **parent** и **child**:



Синхронизация процессов

<u>Системный вызов wait() – приостанавливает выполнение процесса до тех пор, пока не будет завершён другой процесс:</u>

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *status);
```

Системный вызов wait() должен быть вызван исходным процессом столько раз, сколько процессов было им порождено.

Системный вызов wait() возвращает идентификатор завершившегося процесса-потомка. Если status отличен от NULL, то целочисленная переменная, на которую указывает status, будет содержать в старшем байте код завершения процесса-потомка, установленный последним при помощи системного вызова **exit()**, а в младшем – индикатор причины завершения процесса-потомка. Исследовать статус можно при помощи следующих макросов (они принимают в качестве аргумента буфер (типа int), а не указатель на буфер):

WIFEXITED(status)

не равен нулю, если дочерний процесс нормально завершился.

WEXITSTATUS(status)

возвращает восемь младших битов возвращаемого значения завершившегося дочернего процесса. Эти биты могли быть установлены в аргументе функции exit() или в аргументе оператора return в функции main(). Этот макрос можно использовать, только если WIFEXITED вернул ненулевое значение.

Алгоритм wait():

1. Если процесс, который вызвал wait(), не имеет потомков, то возвратить код ошибки.
2. В бесконечном цикле:
   - Если процесс, вызвавший wait(), имеет потомков, прекративших существование:
     - выбрать произвольного потомка;
     - передать его родителю информацию об использовании потомком ресурсов;
     - освободить в таблице процессов место, занятое процессом;
     - выдать идентификатор процесса, код возврата status из системного вызова exit(), вызванного потомком.
   - Приостановиться с приоритетом, допускающим прерывание, до завершения потомка.

Пример использования системного вызова wait():

```
void main (int argc, char* argv[])
{
    int i, ret_val, ret_code;
    for (i = 0; i < 15; i++)
            if (fork() == 0)
            {
                    /*Процесс-потомок*/
                    printf ("процесс-потомок %x\n", getpid());
                    exit (i);
            }
    for (i = 0; i < 15; i++) /*Процесс-родитель*/
    {
            ret_val = wait (&ret_code);
            printf ("wait PID %x with code %x\n", ret_val, ret_code);
    }
}
```

Заранее неизвестно, в какой последовательности потомки завершат выполнение.

Вызов программы в рамках порождённого процесса

Если необходимо выполнить какую-либо сложную операцию, и, что встречается чаще всего, в программе родителя описать эти действия невозможно, то потомок описывается в другом файле, и из родителя, при помощи набора системных вызовов exec(), вызывается другая программа – файл, имя которого указывается первым параметром системного вызова exec().

Запуск программ, находящихся в отдельных файлах, в рамках текущего процесса возможен при использовании следующих системных вызовов:

int execl (char *path, char* arg1, ...);

Запуск программы path с параметрами arg1, arg2… Последний параметр должен иметь значение NULL.

int execv (char *path, char* argv[]);

Запуск программы path с параметрами argv[i]. Последний параметр должен иметь значение NULL.

int execlp (char *file, char *arg1, ...);

Запуск программы file с параметрами arg1, arg2… Последний параметр должен иметь значение NULL. Если file не содержит символов /, поиск программы ведётся, используя переменную окружения PATH.

int execvp (char *file, char* argv[]);

Запуск программы file с параметрами argv[i]. Последний параметр должен иметь значение NULL. Если file не содержит символов /, поиск программы ведётся, используя переменную окружения PATH.

…

Алгоритм exec():

1. Получить описатель файла по имени файла.
2. Проверить, является ли файл исполняемым и имеет ли пользователь право на его использование.
3. Прочитать информацию из заголовка файла, проверив, является ли он загрузочным модулем.
4. Скопировать параметры, переданные функции, в область системного пространства.
5. Для каждой области, присоединённой к процессу:
   - отсоединить соответствующую область.
6. Для каждой области, определённой в загрузочном модуле:
   - выделить соответствующую область;
   - присоединить соответствующую область.
7. Скопировать параметры, переданные функции, в новую область стека задачи.
8. Освободить описатель файла.

Пример использования exec():

```
void main()
{
    int status, ret_val;
    if (fork()) /*запуск программы в рамках исходного процесса*/
    {
            ret_val = wait (&status);
            execl ("/bin/ls", "ls", "-l", 0);
            printf ("ошибка программы ls\n");
    else /*запуск программы в рамках порождённого процесса*/
    {
            execl ("/bin/date", "date", 0);
            printf ("ошибка программы date\n");
    }
    exit (0);
}
```

Средства межпроцессного взаимодействия.

1. Программные каналы.
2. Сигналы.
3. Очередь сообщений.
4. Разделяемая память.
5. Семафоры.
6. Сокеты.

Взаимодействие процессов посредством сигналов.

Сигналы бывают различных видов: сигнал нажатия клавиши на клавиатуре; сигнал истечения времени работы; сигнал, посланный другим процессом…

Сигнал идентифицируется своим номером (некоторое целое число) и именем.

Средства работы с сигналами находятся в библиотеке **signal.h**.

Ядро проверяет получение сигнала процессом в следующих случаях:

- при переходе процесса из режима ядра в режим задачи;
- при переходе процесса в состояние приостанова и при выходе из этого состояния.

Обработка сигнала ядром происходит только в одном случае – при переходе процесса из режима ядра в режим задачи.

Для использования механизма сигналов используются два системных вызова: kill() и signal().

Системный вызов **kill()** – посылка выбранному процессу указанного сигнала.

Посылаемый сигнал может быть обработан процессом-получателем следующим образом:

1. Получение сигнала планируется (ждёт сигнала).
   Выполнение процесса-получателя прерывается, инициализируется запланированная процедура обработки сигнала.
2. Получение сигнала не планируется.
   Процесс-получатель либо заканчивается аварийно, либо останавливается до выполнения определённых условий в зависимости от вида полученного сигнала.
3. Получение сигнала игнорируется.
   Стоит применять во время важной работы, которую нельзя прерывать.

```
int kill (pid_t pid, int sig);
```

pid – процесс или группа процессов, которым посылается сигнал.

pid > 0 – ядро посылает сигнал процессу с указанным идентификатором;

pid = 0 – сигнал посылается всем процессам, входящим в одну группу с процессом, который выполнил системный вызов **kill()**, включая сам процесс, пославший сигнал;

pid = -1 – сигнал посылается всем процессам, у которых код идентификатора пользователя совпадает с тем, под которым выполнялся процесс, вызвавший системный вызов **kill()**;

pid < 0, pid ≠ -1 – сигнал посылается всем процессам, идентификатор группы которых равен модулю первого аргумента ($|pid|$).

Системный вызов **pause()** – приостанавливает выполнение процесса до момента получения любого сигнала.

Пример использования системного вызова kill():

```
#include <signal.h>

void main()
{
    register int i;
    setpgrp();
    for (i = 0; i < 10; i++)
    {
            if (fork() == 0)
            /*порождённый процесс*/
            {
                    if (i & 1) /*нечётные потомки*/
                            setpgrp();
                    printf ("pid=%d pgrp=%d\n", getpid(), getpgrp());
                    pause();
            }
    }
    kill (0, SIGINT);
}
```

Свою работу прекратят только потомки, созданные в чётных итерациях цикла. Процессы, созданные в нечётных итерациях, будут продолжать ждать сигнала.

Системный вызов **signal()** – позволяет задавать режим обработки полученного сигнала отличный от стандартного.

int signal (int sig, void (*func)());

sig – номер сигнала.

func = 0 – при получении сигнала процесс завершит свою работу в режиме ядра

func = 1 – процесс будет игнорировать все последующие получения сигнала с указанным номером.

Любое чётное значение – адрес функции обработки данного сигнала.

Алгоритм обработки сигнала:

1. Выбрать номер сигнала из записи таблицы процессов.
2. Очистить поле с номером сигнала.
3. Если пользователь ранее вызвал функцию **signal()**, с помощью которой сделал указание игнорировать сигнал данного типа:
   - вернуть управление.
4. Если пользователь указал функцию, которую нужно выполнить по получении сигнала:
   - из пространства процессов выбрать виртуальный адрес функции обработки сигнала;
   - **(*)** очистить поле в пространстве процессов с адресом функции обработки сигнала;
   - внести изменения в пользовательский контекст;
   - искусственно создать в стеке задачи запись, имитирующую обращение к функции обработки сигнала;
   - внести изменения в системный контекст;

- записать адрес функции обработки сигнала в поле счётчика команд;
- возвратить управление.

5. Запустить алгоритм **exit()**.

Пункт, помеченный **(*)**, создаёт потенциальную возможность для возникновения нежелательных побочных эффектов. Если процессу вновь потребуется обработать сигнал, ему необходимо будет вновь прибегнуть к выполнению системного вызова **signal()**. Создаются условия для конкуренции, и если второй сигнал поступит до того, как процесс-получатель выполнит системный вызов **signal()**, возникнет неопределённая ситуация – неизвестно, какую функцию вызвать в случае приходы сигнала.

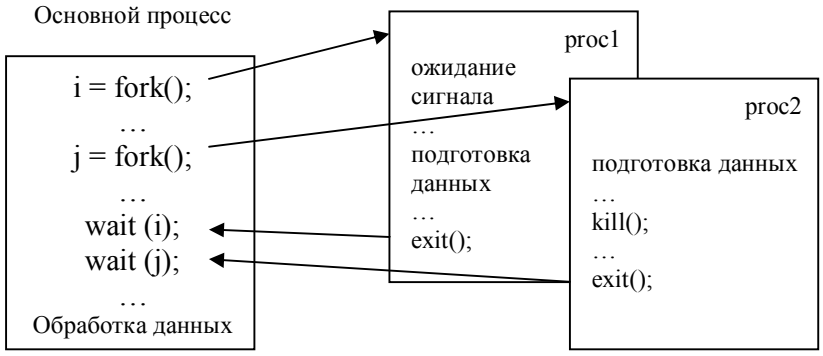
Обработчик сигнала не прерывается другими сигналами.

Пример, иллюстрирующий этот случай:

```
#include <signal.h>

void sigcatcher()
{
    printf ("PID %d принял сигнал\n", getpid());
    signal (SIGINT, sigcatcher);
}

void main()
{
    int ppid;
    signal (SIGINT, sigcatcher);
    if (fork() == 0) /*процесс-потомок*/
    {
            sleep (5); /*задержка 5 секунд*/
            ppid = getppid(); /*идентификатор родителя*/
            for (;;)
                    if (kill (ppid, SIGINT) == -1)
                            exit();
    }
    /*процесс-родитель*/
    nice (10); /*уменьшить приоритет родителя*/
    for (;;);
}
```

Пример взаимодействия процессов:

Данные для основного процесса находятся в порождённых. Основной процесс ожидает завершения работы обоих порождённых процессов, затем начинает обрабатывать данные.



```
#include <signal.h>

void main()
{
    char s[7];
    char *p;
    int status, i, j, k, m;
    if ((i = fork()) == 0)
            execl ("proc1", "proc1", 0);
    if ((j = fork()) == 0)
    {
            m = i;
            s[6] = '\0';
            k = 5;
            while (k != -1)
            {
/*заносим в массив s идентификатор первого порождённого процесса в восьмеричной системе*/
                    s[k] = (m & 07) + '0';
                    m = m >> 3;
                    k--;
            }
/*передаём запускаемой программе в качестве параметра идентификатор первого порождённого процесса*/
            execl ("proc2", "proc2", s, 0);
    }
    if ((k = wait (&status)) != j)
            exit (1);
    if ((k = wait (&status)) != i)
            exit (1);
/*начало обработки данных*/
    ...
}

/*Первый порождённый процесс proc1*/
int ind;

void onittr()
{
    ind++;
}

void main()
{
/*обработка, не зависящая от proc2*/
    signal (10, onittr);
    while (ind == 0)
            sleep (1);
    exit ();
}
```

```
/*Второй порождённый процесс proc2*/

void main (int argc, char* argv[])
{
    int j, c;
    char *p;
    argv++;
    j=0;
    p = argv[0];
    while (c = (*p++))
    {
            if (c >= '0' && c <= '7')
            /*определение номера процесса*/
                    j = j * 8 + c - '0';
            /*подготовка данных*/
            kill (j, 10); /*посылка сигнала*/
            exit();
    }
}
```

Системные операции, связанные со временем.

stime() - позволяет заносить в глобальную переменную, определённую на уровне ядра, системное время в секундах с момента 1 января 1970 года (эта функция доступна только суперпользователю (root)).

time() – выбирает время из глобальной переменной.

times() – суммарное время выполнения процесса и всех его потомков в некоторой структуре данных.

int raise (int signal); – сигнал, указанный аргументом, получает этот же процесс:

int alarm (int secs); – планирует посылку сигнала SIGALARM, определённого в **signal.h**, этому же процессу через указанный промежуток времени. Системный вызов только устанавливает промежуток времени, по истечении которого ядром будет послан сигнал.

Пример:

Процессу посылается сигнал через кратные промежутки времени (1 минута). По получении сигнала, процесс проверяет, имел ли место доступ к файлу в течение последней минуты, и выдаёт соответствующее сообщение.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>

void wakeup()
{
}

void main (int argc, char* argv)
{
    struct stat statbuf;
    time_t axtime;
    axtime = (time_t) 0;
    for (;;)
    {
            /*получение значения времени доступа к файлу*/
            if (stat (argv[1], &statbuf) == -1)
            {
                    printf ("файла с именем %s нет\n", argv[1]);
                    exit (0);
            }
            if (axtime != statbuf.st_atime)
            {
                    printf ("к файлу %s было обращение\n", argv[1]);
                    axtime = statbuf.st_atime;
            }
            signal(SIGALARM,wakeup);
            /*распоряжение принимать сигнал*/
            alarm (60);
            pause (); /*ждать до получения сигнала*/
    }
}
```

Командный процессор shell.

Некоторые команды shell обрабатывает своими средствами (внутренние команды), некоторые – внешними (внешние команды). При обработке команд внешними средствами порождается процесс, в рамках которого запускается программа. В случае необходимости процессы связываются при помощи каналов.

Рассмотрим алгоритм работы shell:

На входе – строка ls –la | wc >a.txt [&]

```
/*чтение командной строки до символа конца строки*/
while (read (stdin, buffer, numchars))
{
    /*синтаксический разбор командной строки*/
    if (/*командная строка содержит &*/)
            amper = 1;
    else
            amper = 0;
    /*для команд, не являющихся конструкциями командного языка Shell*/
    if (fork() == 0) /*дочерний процесс*/
    {
            if (/*переадресация ввода-вывода*/)
            {
                    fd = creat (newfile, fmsk);
                    close (stdout);
                    dup (fd);
                    close (fd);
            }
            if (/*используются каналы?*/)
            {
                    pipe (filedes);
                    if (fork() == 0) /*внучатый процесс*/
                    {
                            /*первая компонента командной строки (ls –la)*/
                            close (stdout);
                            dup (fildes[1]);
                            close (fildes[0]);
                            close (fildes[1]);
                            /*стандартный вывод направлен в канал*/
                            execv (command1, command1, 0);
                    }
                    /*вторая компонента командной строки (wc)*/
                    close (stdin);
                    dup (fildes[0]);
                    close (fildes[0]);
                    close (fildes[1]);
                    /*стандартный ввод будет вестись из канала*/
            }
            execv (command2, command2, 0);
    }
    /*родительский процесс*/
/*процесс-родитель ждёт завершения потомка, если это вытекает из введённой строки*/
    if (amper == 0)
            read = wait (&status);
}
```

Механизмы межпроцессного взаимодействия.

Общим пользовательским механизмом взаимодействия процессов, в общем случае несвязанных отношением родства, является набор системных средств IPC (Inter-Process Communication Facilities).

В этот набор средств входят:

1. Средства, обеспечивающие возможность синхронизации процессов при доступе к разделяемому ресурсу:
   - семафоры;
   - блокировки чтения-записи.
2. Средства, обеспечивающие возможность посылки процессом сообщения другому (в общем случае – произвольному) процессу:
   - каналы;
   - очереди сообщений (message queues).
3. Средства, обеспечивающие наличие общей для процессов памяти:
   - сегменты разделяемой памяти (shared memory segments).
4. Средства удалённого вызова процедур RPC (remote procedure call), позволяющие взаимодействовать процессам в сети.

Основы теории семафоров.

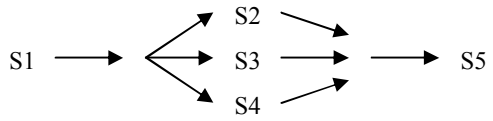
(теория голландского математика Е. Дейкстры)

1. Простой пример

Ставится задача написать программную реализацию проблемы взаимного исключения. Два процесса: <процесс 1> и <процесс 2>, для удобства циклические. В каждом цикле процесса существует критический интервал, критический в том смысле, что в каждый момент времени только один из процессов может находиться внутри своего критического интервала. Для реализации такого взаимного исключения будем считать, что оба процесса имеют доступ к некоторому числу общих переменных. Считаем, что проверка текущего значения общей переменной и присваивание нового значения есть неделимая операция: если оба процесса <одновременно> меняют значение общей переменной, то фактически это происходит последовательно; если процесс проверяет значение переменной одновременно с присваиванием ей значения другим процессом, то первый обнаружит либо старое, либо новое значение, но не их смесь.

Конструкция parbegin parend будет означать параллельный блок с параллельным выполнением составляющих конструкцию операторов.

{ S1; parbegin S2; S3; S4; parend S5 }



Первый вариант решения проблемы взаимного исключения может быть следующим:

```
{
    int очередь = 1;                ← общая переменная
    parbegin
    процесс 1:
    {
            L1:
            if (очередь == 2)
                    goto L1;
            критический интервал 1;
            очередь = 2;
            остаток цикла 1;
            goto L1;
    }
    процесс 2:
    {
            L2:
            if (очередь == 1)
                    goto L2;
            очередь = 1;
            остаток цикла 2;
            goto L2;
    }
    parend
}
```

Недостатки: после завершения критического интервала 1 первого процесса переменная очередь принимает значение 2, что не процессу повторно войти в критический интервал до тех пор, пока эта переменная не получит значения 1 во втором процессе. Как результат этого, процессы должны быть синхронизированы: 1, 2, 1, 2, 1, 2, …

Второй вариант решения:

```
{
    int c1, c2;
    parbegin
    процесс 1:
    {
            L1:
            if (c2 == 0)            ← процесс 2 в критическом интервале
                    goto L1;
            c1 = 0;
            критический интервал 1;
            c1 = 1;
            остаток цикла 1;
            goto L1;
    }
    процесс 2:
    {
            L2:
            if (c1 == 0)            ← процесс 1 в критическом интервале
                    goto L2;
            c2 = 0;
            критический интервал 2;
            c2 = 1;
            остаток цикла 2;
            goto L2;
    }
    parend
}
```

Недостатки: полностью взаимное исключение не гарантировано, т.к. возможна ситуация: процесс 1 обнаруживает, что c2 == 1, а вслед за этим процесс 2 немедленно проверяет, что c1 == 1, и оба, убедившись, что другой процесс не находится в критическом интервале, входят в свой критический интервал.

Третий вариант решения (меняем местами установку своего индикатора и проверку чужого):

```
{
    int c1, c2;
    c1 = 1; c2 = 1;
    parbegin
    процесс 1:
    {
            A1:
            c1 = 0;
            L1:
            if (c2 == 0)            ← процесс 2 в критическом интервале
                    goto L1;
            критический интервал 1;
            c1 = 1;
            остаток цикла 1;
            goto A1;
    }
    процесс 2:
    {
            A2:
            c2 = 0;
            L2:
            if (c1 == 0)            ← процесс 1 в критическом интервале
                    goto L2;
            критический интервал 2;
            c2 = 1;
            остаток цикла 2;
            goto L2;
    }
    parend
}
```

Решение полностью безопасно с точки зрения взаимного исключения, но зато появляется опасность взаимной блокировки: если процесс 1 выполнит c1 = 0, но еще до проверки им c2 процесс 2 выполнит c2=0, оба процесса заблокируют друг друга.

Четвертый вариант решения (обратная установка собственного индикатора в 1 при неудачной попытке входа в критический интервал):

```
{
    int c1, c2;
    c1 = 1;
    c2 = 1;
    parbegin
    процесс 1:
    {
            L1:
            c1 = 0;
            if (c2 == 0)            ← процесс 2 в критическом интервале
            {
                    c1 = 1;         ← процесс установит свой индикатор в 1 при неудачной
                    goto L1;        попытке входа в критический интервал
            }
            критический интервал 1;
            c1 = 1;
            остаток цикла 1;
            goto A1;
    }
    процесс 2:
    {
            L2:
            c2 = 0;
            if (c1 == 0)
            {
                    c2 = 1;
                    goto L2;
            }
            критический интервал 2;
            c2 = 1;
            остаток цикла 2;
            goto L2;
    }
    parend
}
```

Решение столь же безопасное, как и предыдущее + <одновременное> присваивание c1 = 0 и c2 = 0 не приводит к взаимоблокировке, т. к. каждый процесс установит свой индикатор (c1, c2) в 1 при неудачной попытке входа в критический интервал. Однако решение также неудовлетворительно, поскольку можно подобрать такие скорости, что процессы будут проверять чужие <c> только в те моменты, когда значения их собственных <c> нулевые. (Аналогия с действиями участников телефонного разговора после обрыва связи - в этом плане решение достаточно приемлемо в повседневной жизни).

Решение Деккера (голландский математик):

```
{
    int c1, c2, очередь;
    c1 = 1; c2 = 1; очередь = 1;
    parbegin
    процесс 1:
    {
            A1:
            c1 = 0;
            L1:
            if (c2 == 0)
            {
                    if (очередь == 1)
                            goto L1;
                    c1 = 1;
                    B1:
                    if (очередь == 2)
                            goto B1;
                    goto A1;
            }
            критический интервал 1;
            очередь = 2;
            c1 = 1;
            остаток цикла 1;
            goto A1;
    }

процесс 2:
    {
            A2:
            c2 = 0;
            L2:
            if (c1 == 0)
            {
                    if (очередь == 2)
                            goto L2;
                    c2 = 1;
                    B2:
                    if (очередь == 1)
                            goto B2;
                    goto A2;
            }
            критический интервал 2;
            очередь = 1;
            c2 = 1;
            остаток цикла 2;
            goto A2;
    }
    parend
}
```

Убедитесь самостоятельно в корректности решения.

Выводы:

- недостаточность обычных программных средств для решения простейшей задачи взаимного исключения;
- громоздкое описание отдельных процессов.

Иначе говоря, предыдущий набор средств связи между процессами является неадекватным для рассматриваемой проблемы!

2. Синхронизирующие примитивы

Новые средства:

- специальные целочисленные неотрицательные переменные, названные "семафорами";
- две элементарные операции, названные "P-операция" и "V-операция".

V-операция, аргументом которой является семафор, выполняет увеличение значения аргумента на 1, причем действие рассматривается как неделимая операция.

Замечание: V(S1) ≠ S1=S1+1

P-операция, аргументом которой является семафор, выполняет уменьшение значения аргумента на 1, если только результат не становится отрицательным, причем действие рассматривается как неделимая операция.

Если значения семафора равны 0 или 1 – семафор двоичный (достаточно для решения задачи взаимного исключения), иначе семафор называется общим.

В результате задача взаимного исключения становится тривиальной даже в случае нескольких процессов:

```
{
    int свободно;
    свободно = 1;
    parbegin
            процесс 1: {…}
            процесс 2: {…}
            …
            процесс n: {…}
    parend
}
```

где i-й процесс имеет вид:

```
процесс i:
{
    Li:
    P (свободно);
    критический интервал i;
    V (свободно);
    остаток цикла i;
    goto Li;
}
```

3. Использование семафоров (алгоритм банкира)

ОС (банкир) обладает конечным числом страниц (конечным капиталом в $). Она (он) обслуживает пользовательские процессы (клиентов), которые могут занимать у нее (у него) страницы ($) на следующих условиях:

1. Пользовательский процесс (клиент) делает запрос (заем) для выполнения некоторой работы (для совершения сделки), которая будет завершена за конечный промежуток времени.

2. Пользовательский процесс (клиент) должен заранее указать максимальную потребность в страницах (в $) для выполнения работы (для совершения сделки).

3. Пока запрос (заем) не превышает заранее установленную потребность, пользовательский процесс (клиент) может увеличивать или уменьшать свой заем страницу за страницей ($ за $).

4. Пользовательский процесс (клиент), который просит увеличить его текущий заем, обязуется принимать без недовольства следующий ответ: <Если бы я дал вам запрашиваемую страницу ($), вы еще не превысили бы свою установленную потребность, и поэтому вы имеете право на следующую страницу ($). Однако в настоящий мне, по

некоторым причинам, неудобно ее (его) дать, но я обещаю вам страницу ($) в должное время>.

5. Гарантия для пользовательского процесса (клиента) в том, что этот момент действительно наступит, основана на предусмотрительности ОС (банкира) и на том факте, что остальные пользовательские процессы (клиенты) подчиняются тем же условиям, что и он сам: как только пользовательский процесс (клиент) получает страницу ($), он приступает к своей работе (к своей сделке) с ненулевой скоростью, т.е. в конечный промежуток времени он или запросит новую страницу (новый $), или возвратит страницу ($), или закончит работу (сделку), что означает возвращение всех страниц (всего займа).

Основными вопросами являются:

а) при каких услових ОС (банкир) может приступить к обслуживанию нового пользовательского процесса (заключить контракт с новым клиентом)?

б) при каких услових ОС (банкир) может выдать следующую страницу ($) пользовательскому процессу (клиенту), не опасаясь попасть в тупиковую ситуацию? (российский аналог краха Кредо-банка, СТБ).

Ответ на вопрос а) прост: ОС (банкир) может приступить к обслуживанию любого пользовательского процесса (клиентом), чья максимальная потребность в страницах ($) не превышает буферного пула страниц (капитала банкира).

Ответ на вопрос б) эквивалентен вопросу, могут ли быть с гарантией завершены все обслуживания процессов (все сделки).

Кроме этого, имеется ряд предположений для разрабатываемой программы.

Предположим, что пользовательские процессы пронумерованы от 1 до N, и что страницы пронумерованы от 1 до M. С каждым процессом связана переменная <номер_страницы>, значение которой после очередной выдачи страницы определяет номер только что выданной страницы. В свою очередь с каждой страницей связана переменная <номер_процесса>, значение которой указывает процесс, которому была выдана.

Каждый пользовательский процесс имеет переменную состояния <процесс_пер> - переменная клиента; при этом процесс_пер=1 означает <Мне нужна страница> (иначе процесс_пер=0). Каждая страница имеет переменную состояния <страница_пер>; при этом страница_пер=1 означает, что она находится среди буферного пула свободных страниц. С каждым процессом и каждой страницей связаны двоичные семафоры <процесс_сем> и <страница_сем>.

Предполагается, что каждая страница выдается и возвращается на основании указаний пользовательского процесса, но процесс не имеет возможности возвратить взятую страницу мгновенно. После того, как процесс заявляет, что страница ему больше не нужна, последняя не становится немедленно доступной для последующего использования. Фактически возврат страницы заканчивается после того, как та действительно присоединится к буферному пулу свободных страниц: об этом страница будет сообщать процессу с помощью общего семафора процесса <возвращенные_страницы>. Р-операция над этим семафором должна предостеречь пользовательский процесс от неумышленного превышения своей максимальной потребности. Перед каждым запросом страницы процесс выполнит Р-операцию над семафором <возвращенные_страницы>, а начальное значение переменной <возвращенные_страницы> полагается равным потребности.

```
int заем[N], требование[N], процесс_сем[N], процесс_пер[N], номер_страницы[N],
    возвращенные_страницы[N], страница_сем[l], страница_пер[l], номер_процесса[l];
int взаимоискл, V_буфера, k;

boolean procedure попытка_выдать_страницу_процессу (int j)
{
    int i, свободные_страницы;
    boolean завершение_под_сомнением[N];
    if (процесс_пер[j] == 1)
    {
            свободные_страницы = V_буфера – 1;
            требование[j] = требование[j] – 1;
            заем[j] = заем[j] + 1;
            for (i = 0; i <= N; i++)
                завершение_под_сомнением[i] = true;
L0:         for (i = 1; i <=N; i++)
            {
                /*Проверка на безопасность завершения*/
                if (завершение_под_сомнением[i] and требование[i] <= свободные_деньги)
                {
                    if (i ≠ j)
                    {
                        завершение_под_сомнением[i] = false;
                        свободные_деньги = свободные_деньги + заем[i];
                        goto L0;
                    }
                    else
                    {
                        i = 0;
L1:                     i = i + 1;
                        if (страница_пер[i] == 0)
                            goto L1;
                        номер_страницы[j] = i;
                        номер_процесса[i] = j;
                        процесс_пер[j] = 0;
                        страница_пер[i] = 0;
                        V_буфера = V_буфера – 1;
                        попытка_выдать_страницу_процессу = true;
                        V (процесс_сем{j]);
                        V (страница_сем[i]);                    ← *
                        return;
                    }
                }
            }
            требование[j] = требование[j] + 1;
            заем [j] = заем[j] – 1;
    }
    попытка_выдать_страницу_процессу = false;
}

взаимоискл = 1;
V_буфера = M;
for (k = 1; k <= N; k++)
{
    заем[k] = процесс_сем[k] = процесс_пер[k] = 0;
    требование[k] = потребность[k];
    возвращенные_страницы[k] = потребность[k];
}
```

У процесса запрос новой страницы (часть программы процесса) описывается след. последовательностью операторов:

```
P (возвращенные_страницы[n]);
P (взаимоискл);
процесс_пер[n] = 1;
попытка_выдать_страницу_процессу (n);
V (взаимоискл);
P (процесс_сем[n]);
```

После завершения последнего оператора значение переменной <номер страницы[n]> определяет только что выданную страницу; процесс может использовать его, но обязан возвратить его в надлежащее время ОС.

Структура программы для страницы:

```
Страница m:
{
    int h;
L:  P (страница_сем[m]);                ← * (разблокирует процесс)
/*теперь номер_процесса[m] указывает процесс, которому выдана страница.
Страница обслуживает процесс до тех пор, пока она необходима.
Для возвращения в буферный пул страница поступает следующим образом:*/
    P (взаимоискл);
    требование[номер_процесса[m]] = требование[номер_процесса[m]] + 1;
    заем[номер_процесса[m]] = заем[номер_процесса[m]] – 1;
    страница_пер[m] = 1;
    V_буфера = V_буфера +1;
    V (возвращенные_страницы[номер_процесса[m]]);     ← Сообщение процессу
J:  V (взаимоискл);
    goto L;
}
```

Общие понятия средств IPC.

Классы UNIX-систем:

1. POSIX.
2. System V.

Каждый из этих классов систем содержат средства IPC, однако эти средства существенно отличаются. Мы будем в дальнейшем рассматривать средства IPC для System V.

1. Понятие ключа.

Ключ (key) – число, идентифицирующее объект межпроцессорного взаимодействия. Ключ существует для любого средства IPC: очереди сообщений, набора семафоров, сегмента разделяемой памяти, …

Тип ключа – key_t, состав зависит от реализации для различных ОС; определён в файле <sys/types.h>

Для того, чтобы два или более процесса могли взаимодействовать между собой, необходим договор между ними об уникальности идентификации ключа используемых ресурсов.

Создание ключа:

а) программно обеспечить уникальный ключ объекта IPC;

б) использовать системный вызов ftok();

```
key_t ftok (const char *pathname, int ind);
```

pathname – имя файла;

ind – идентификатор проекта (младшие 8 бит не должны быть нулевыми).

- вызывается системный вызов stat();

- объединяется информация о файловой системе (поле st_dev структуры stat), номер inode (поле st_ino), 8 младших бит идентификатора ind.

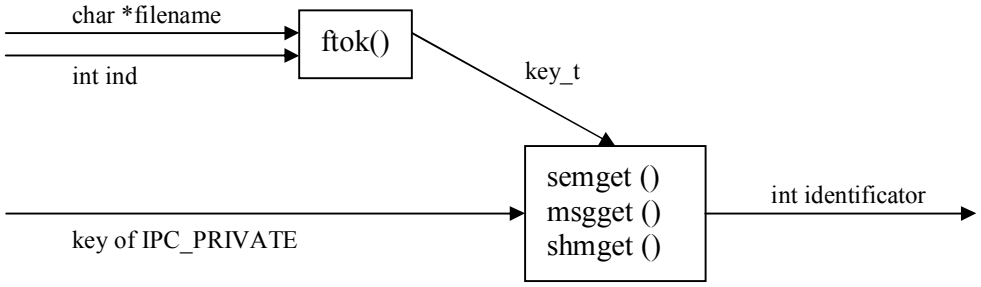
в) указать в качестве ключа константу вида IPC_PRIVATE.

2. Ключ используется как для создания объекта межпроцессного взаимодействия, так и для доступа к существующему объекту. Обе операции выполняются с помощью набора системных вызовов get():

semget() – получение семафора;
msgget() – получение очереди сообщений;
shmget() – получение сегмента разделяемой памяти.

Возвращает идентификатор, который можно использовать в последующих действиях.

Если сравнивать объекты IPC с файлами, то ключ соответствует имени файла, а идентификатор – дескриптору файла.



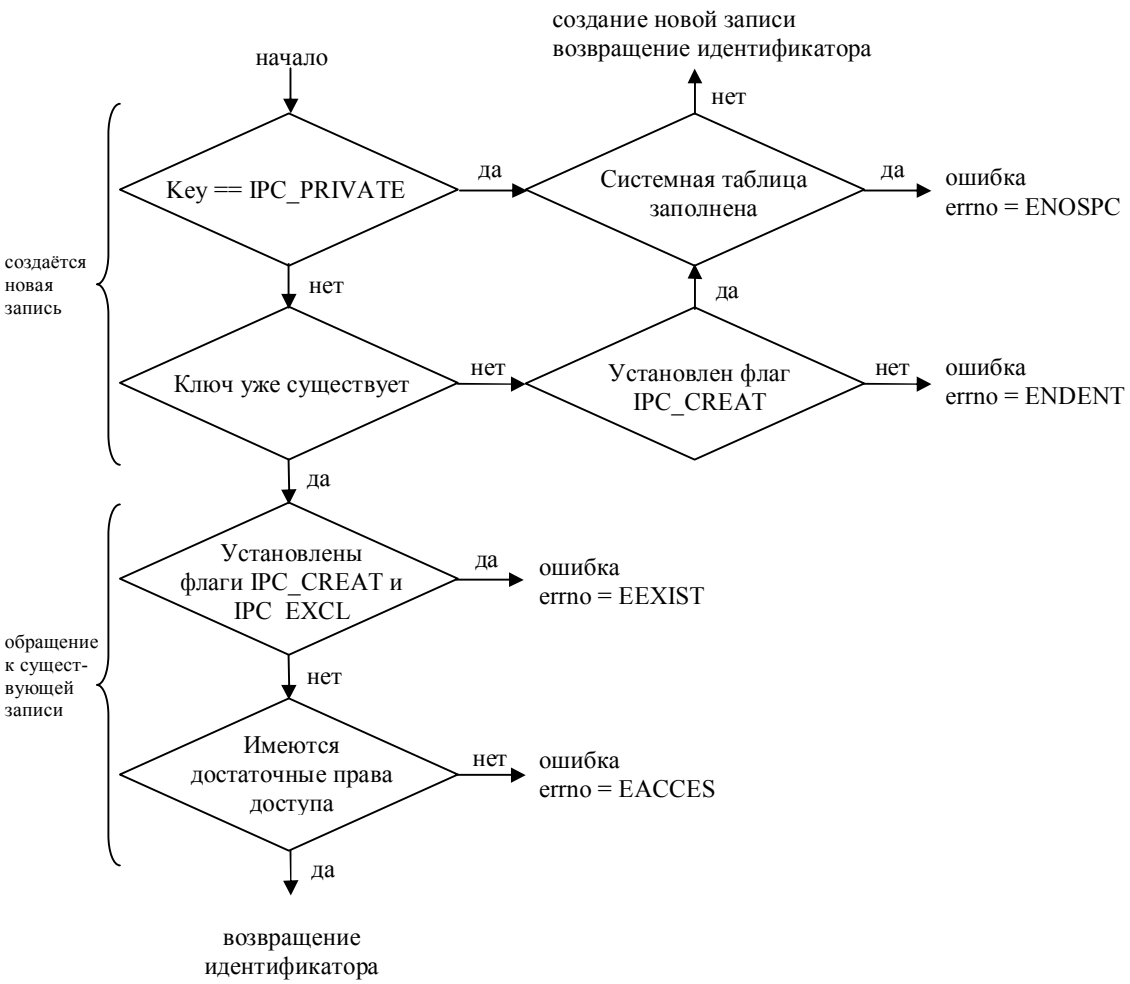
Создание или открытие объекта IPC этими функциями управляется двумя флагами:

- IPC_CREAT

Любая из перечисленных функций создаёт новый объект IPC для данного ключа. Если флаг IPC_CREAT не задан, а объект IPC с указанным ключом существует, то обращающийся процесс получит идентификатор существующего объекта.

- IPC_EXCL

Указанный флаг совместим с предыдущим флагом. Он предназначен для создания объекта IPC. Если объект IPC уже существует, то любая из этих функций возвратит -1, а системная переменная errno будет содержать значение EEXIST.



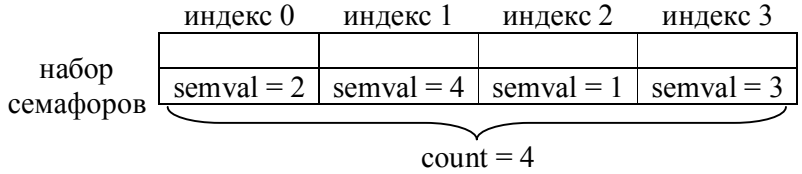
3. Для всех средств межпроцессорного взаимодействия имеются схожие функции управления, которые позволяют опросить объекты IPC и изменить их статус.

4. Для каждого объекта IPC создаётся структура статуса IPC, которая имеет общую структуру в части прав доступа, а именно:

…
uid_t cuid; – идентификатор пользователя, который создал объект
gid_t cgid; – идентификатор группы, которой принадлежал пользователь, создавший объект
uid_t uid; – действующий идентификатор пользователя
gid_t gid; – действующий идентификатор группы
mode_t mode; – права доступа
…

Синхронизация на основе семафоров.

Средства IPC расширяют понятие семафора до набора семафоров.



Основной смысл введения набора семафоров – то, что, независимо от количества семафоров, каждая операция над набором семафоров – атомарная операция.

Действие вида семафор = семафор +1 и увеличение значения семафора – разные вещи.
Семафор определяется:
- значением семафора;
- идентификатором процесса, который последним работал с семафором;
- число процессов, ожидающих увеличения значения семафора;
- число процессов, ожидающих нулевого значения семафора.

Системные вызовы для работы с семафорами:
semget() – применяется для создания или получения доступа к набору семафоров
semop() – манипуляция значением семафоров
semctl() – выполнение управляющих действий над набором семафоров

Системный вызов semget():
```
semid = int semget (key_t key, int count, int flag);
```

semid – идентификатор набора семафоров
key – ключ набора семафоров
count – число семафоров в наборе
flag – дополнительные флаги (определены в файле **<sys/ipc.h>**):
9 младших бит флага используются для создания прав доступа к семафору.

Пример:
```
mqid = semget ((key_t) 0100, 2, 0600 | IPC_CREAT | IPC_EXCL);
```

Системный вызов semctl():
```
int semctl (int semid, int sem_num, int command, union semun arg);
```

sem_num – номер семафора в наборе
command – код операции
Четвёртый параметр задаётся в зависимости от операции. Ранее в качестве четвертого параметра использовалось объединение semun, в последних версиях данная структура исключена из заголовочных файлов. Пользователь может сам определить данную структуру:

```
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
}
```

Операции, которые можно выполнить над набором семафоров:
IPC_RMID – уничтожить набор семафоров или одного семафора в наборе;
GETVAL (GETALL) – вернуть значение отдельного семафора (всех семафоров в группе)
SETVAL (SETALL) – установить значение отдельного семафора (всех семафоров в группе)
GETPID – вернуть число семафоров в группе

Пример инициализации семафоров:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

int initsem (key_t semkey) /*для инициализации семафора*/
{
    int status = 0, semid;
    union semun arg;
    if ((semid = semget (semkey, 1, 0600 | IPC_CREAT | IPC_EXCL)) == -1)
    {
            if (errno == EEXIST) /*если набор семафоров существовал ранее*/
                    semid = semget (semkey, 1, 0); /*то открыть его*/
            else /*семафор только что создан, инициализировать его*/
            {
                    arg.val = 1; /*начальное значение семафора*/
                    status = semctl (semid, 0, SETVAL, arg);
            }
    }
    if (semid == -1 || status == -1)
            return -1;
    return semid;
}
```

Системный вызов semop():

int semop (int semid, struct sembuf *op_array, size_t count);

semid – идентификатор набора семафоров

op_array – некоторый массив структур, определённых в файле **<sys/sem.h>**

count – размерность этого массива

Возвращаемое значение – значение последнего обработанного семафора.

Каждый элемент массива op_array[] имеет следующую структуру:

```
struct sembuf
{
    unsigned sem_num;
    short sem_op;
    short sem_flg;
};
```

sem_num – номер семафора в наборе;

sem_op – операция над семафором;

sem_flg – некоторые флаги. Если вам не нужен ни один флаг, то необходимо задать sem_flg равным нулю, иначе он инициализируется случайным образом.

Операции:

1. Отрицательное значение sem_op.

Если sem_op по абсолютному значению не больше значения семафора, то ядро прибавляет это отрицательное значение к значению семафора (уменьшает значение) – *операция P()*.

Если в результате значение семафора стало равным нулю, ядро активизирует все процессы, ожидающие нулевого значения этого семафора.

Если sem_op по абсолютному значению больше значения семафора, то ядро увеличивает на 1 число процессов, ожидающих увеличения значения семафора, и данный процесс переводится в состояние ожидания до наступления указанного события.

2. Положительное значение sem_op.

Значение sem_op добавляется к значению семафора, и все процессы, ожидающие увеличения значения семафора, пробуждаются – *операция V()*.

3. Нулевое значение sem_op.

Если значение семафора равно нулю, выбирается следующий элемент из массива op_array.

Если значение семафора отлично от нуля, ядро увеличивает на 1 число процессов, ожидающих нулевого значения семафора, а процесс, который вызвал этот системный вызов, переводится в состояние ожидания.

Флаг IPC_NOWAIT – при его установке ядро не блокирует текущий процесс, а лишь сообщает в переменной errno о возникновении ситуации, которая могла бы привести к блокировке в случае отсутствия этого параметра. Системный вызов **semop()** в этом случае вернет -1.

Пример функций P и V для реализации взаимного исключения:

```
int p (int semid)
{
    struct sembuf p_buf;
    p_buf.sem_num = 0;
    p_buf.sem_op = -1;
    p_buf.sem_flg = 0;
    if (semop (semid, &p_buf, 1) == -1)
            exit (1);
    return 0;
}

int v (int semid)
{
    struct sembuf p_buf;
    p_buf.sem_num=0;
    p_buf.sem_op = 1;
    p_buf.sem_flg = SEM_UNDO;
    if (semop (semid, &p_buf, 1) == -1)
            exit (1);
    return 0;
}

void main()
{
    key_t semkey = 0x200;
    int i, semid, ret_code, status;
    pid_t pid;
    union semun arg;
    for (i = 0; i < 3; i++)
            if (fork() == 0)
            {
                    pid = getpid();
                    if ((semid = initsem (semkey)) < 0)
                            exit (1);
                    printf ("Процесс %d перед критическим участком\n", pid);
                    p (semid);
                    printf ("Процесс %d выполняет критический участок\n", pid);
                    /*нечто осмысленное в программе*/
                    printf ("Процесс %d покинул критический участок\n", pid);
                    v (semid);
                    printf ("Процесс %d завершал работу\n", pid);
                    exit(0);
            }
    for (i = 0; i < 3; i++)
            wait (&ret_code);
    printf ("Процесс-родитель завершился\n");
    semid = initsem (semkey);
    status = semctl (semid, 0, IPC_RMID, arg);
    if (status == -1)
            printf ("Ошибка закрытия семафора\n");
    else
            printf ("Закрытие семафора\n");
}
```

Передача сообщений между процессами.

Инструменты передачи сообщений между процессами:

1. Каналы.
2. Очереди сообщений IPC.

pipe() – канал для односторонней связи. Можно ли при помощи pipe() организовать двустороннюю связь?

Случай 1:

child:
(1) read через fd[0]
(2) write через fd[1]
exit()

parent:
(3) read через fd[0]
(4) write через fd[1]
exit()

Возможные варианты действий:

I. (3) parent → "p"
(4) parent ← "p"
Потомок никогда не завершится

II. (1) – канал пуст, ожидание
(3) parent → "p"
(1) child ← "p"
(2) child → "c"
(4) parent ← "c"
Успешный обмен данными и завершение работы как родителя, так и потомка

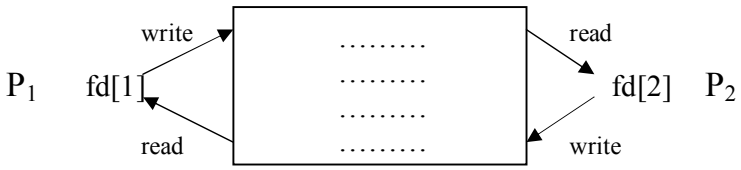
Теперь добавим в действия родителя ожидание завершения потомка.

parent:
(3) read через fd[0]
(4) write через fd[1]
(5) wait()
exit()

В таком случае вариант I преобразуется в вариант III:

III. (3) parent → "p"
(4) parent ← "p"
Оба процесса никогда не завершатся.

Случай 2:



child:
read через fd[1]
write через fd[1] → "c"
exit()

parent:
write через fd[0] → "p"
read через fd[0]
exit()

В этом случае процессы гарантированно обменяются сообщениями, *однако так делать нельзя!* Для нормального обмена сообщениями между процессами нужно создавать два канала.

Очереди сообщений IPC.

Системные вызовы для работы с очередями сообщений:

1. int msgqid = msgget (key_t key, int flag) – создание или открытие уже существующей очереди сообщений.
   msgqid – идентификатор очереди сообщений;
   key – уникальный ключ объекта IPC;
   flag – некоторый набор флагов (аналогичен флагам системного вызова semget).
2. int msgsnd (int msgqid, void *msg, size_t size, int flag) – помещение сообщения в очередь.
   msg – указатель на структуру вида:

```
struct msg
{
        long mtype; //тип сообщения
        char mtext[size]; //текст сообщения
}
```

   size – размер структуры **msg**;
   flag – действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти (флаг IPC_NOWAIT).

Условиями успешной постановки сообщения в очередь являются:

- наличие прав процесса по записи в данную очередь сообщений;
- не превышение длиной сообщения заданного системой верхнего предела;
- положительное значение указанного в сообщении типа сообщения.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в данной очереди сообщений (флаг IPC_NOWAIT при этом отсутствует), то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений.

3. int msgrcv (int msgqid, void *msg, size_t size, long msg_type, int flag) – приём сообщения.
   msg_type – тип принимаемого сообщения;
   остальные параметры аналогичны параметрам системного вызова **msgsnd**.
4. int msgctl (int msgqid, int command, struct msqid_ds *msg_stat) – выполнение управляющих действий над очередью сообщений.
   command – некоторая команда:

IPC_STAT – опроса состояния описателя очереди сообщений и помещения его в структуру **msg_stat**.

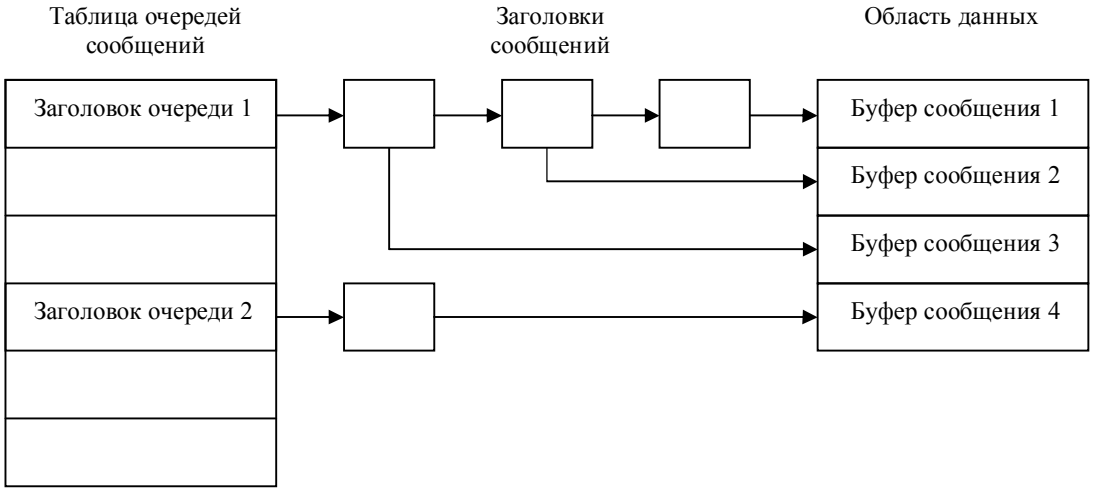
IPC_SET – изменение состояния очереди сообщений, например, изменение прав доступа.

IPC_RMID – уничтожение указанной очереди сообщений.

Структура **msqid_ds** имеет вид:

```
struct msqid_ds
{
        struct ipc_perm msg_perm;  /*структура, описывающая общие для объектов IPC поля данных: ключ, права доступа…*/
        __time_t msg_stime;        /*время последнего выполнения msgsnd*/
        unsigned long int __unused1;
        __time_t msg_rtime;        /*время последнего выполнения msgrcv*/
        unsigned long int __unused2;
        __time_t msg_ctime;        /*время последнего выполнения msgctl*/
        unsigned long int __unused3;
        unsigned long int __msg_cbytes; /*текущее количество байт в очереди*/
        msgqnum_t msg_qnum;        /*текущее количество сообщений в очереди*/
        msglen_t msg_qbytes;       /*максимальное количество байт, доступное для помещения в очередь*/
        __pid_t msg_lspid;         /*идентификатор процесса, который последний отправил сообщение*/
        __pid_t msg_lrpid;         /* идентификатор процесса, который последний принял сообщение */
        unsigned long int __unused4;
        unsigned long int __unused5;
};
```

При построении очереди создаётся также таблица очередей сообщений:



В заголовке содержатся указатели на первое и последнее сообщение в данной очереди, число сообщений и общее количество байт, занимаемое этими сообщениями, идентификаторы процессов, которые последними отправили или приняли через данную очередь, временные метки последних выполненных операций (**msgsnd()**, **msgrcv()**, **msgctl()**).

Процесс обращается к операции **msgsnd(),** проверяет, можно ли поместить сообщение в очередь и не превышен ли максимальный объём сообщения.

После того, как системный вызов **msgsnd()** помещает сообщение в очередь, активизируются все процессы, ожидающие получения из данной очереди сообщения с указанным типом.

Если в очереди сообщений нет сообщения с типом, заданным в **msgrcv()**, то процесс засыпает.

При получении сообщения:

- Проверяются полномочия процесса на чтение из указанной очереди.
- Если в качестве аргумента типа читаемого сообщения указан 0, читается любое первое сообщение из очереди.
- Если тип получаемого сообщения указан отличным от нуля, читается первое сообщение с данным типом.
- Если значение аргумента, задающего тип получаемого сообщения, отрицательно – из очереди выбирается первое сообщение, значение типа которого не более абсолютного значения аргумента.
- Если в указанной очереди отсутствует сообщение с выбираемым типом, то процесс переходит в состояние ожидания до появления нужного сообщения.

Разделяемая память.

Системные вызовы для работы с разделяемой памятью:

1. int shmid = shmget (key_t key, size_t size, int flag) – создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом.

   shmid – идентификатор сегмента разделяемой памяти;

   key – уникальный ключ объекта IPC;

   size – желаемый размер сегмента в байтах.

   Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является идентификатор существующего сегмента. В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера.

   Флаги аналогичны флагам системного вызова **semget()**.
2. void *virtaddr = shmat (int shmid, void *daddr, int flags) – подключение сегмента к виртуальной памяти.

   daddr – желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти;

   virtaddr – фактический виртуальный адрес начала сегмента.

   Если значением daddr является NULL, ядро выбирает наиболее удобный виртуальный адрес начала сегмента.

   flags – флаги:

   SHM_RDONLY – подключение участка памяти только для чтения.

   SHM_RND - определяет, если возможно, способ обработки ненулевого значения daddr.
3. int shmdt (*daddr) – отключение сегмента от виртуальной памяти.

daddr – виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова **shmat()**.

4. int shmctl (int shmid, int command, struct shmid_ds *shm_stat) – полностью аналогичен **msgctl()**.

Структура **shmid_ds** имеет вид:

```
struct shmid_ds
{
        struct ipc_perm shm_perm;       /*структура, описывающая общие для объектов IPC поля данных: ключ, права доступа…*/
        size_t shm_segsz;               /*размер сегмента в байтах*/
        __time_t shm_atime;             /* время последнего выполнения shmat()*/
        unsigned long int __unused1;
        __time_t shm_dtime;             /*время последнего выполнения shmdt()*/
        unsigned long int __unused2;
        __time_t shm_ctime;             /*время последнего изменения при помощи shmctl()*/
        unsigned long int __unused3;
        __pid_t shm_cpid;               /*идентификатор процесса-создателя*/
        __pid_t shm_lpid;               /*идентификатор процесса, последним выполнившего какие-либо операции с текущим сегментом*/
        shmatt_t shm_nattch;            /*количество текущих подключений*/
        unsigned long int __unused4;
        unsigned long int __unused5;
};
```

Выполнение **shmget()** не означает немедленного выделения под него оперативной памяти. Присоединение сегмента к оперативной памяти выполняется системным вызовом shmat(), при этом пользователь получает адрес начала сегмента.

Уничтожение сегмента:

Если на момент выполнения **shmctl()** к сегменту не подключился ни один процесс, то память, занятая сегментом, освобождается, и сегмент разделяемой памяти как объект IPC удаляется.

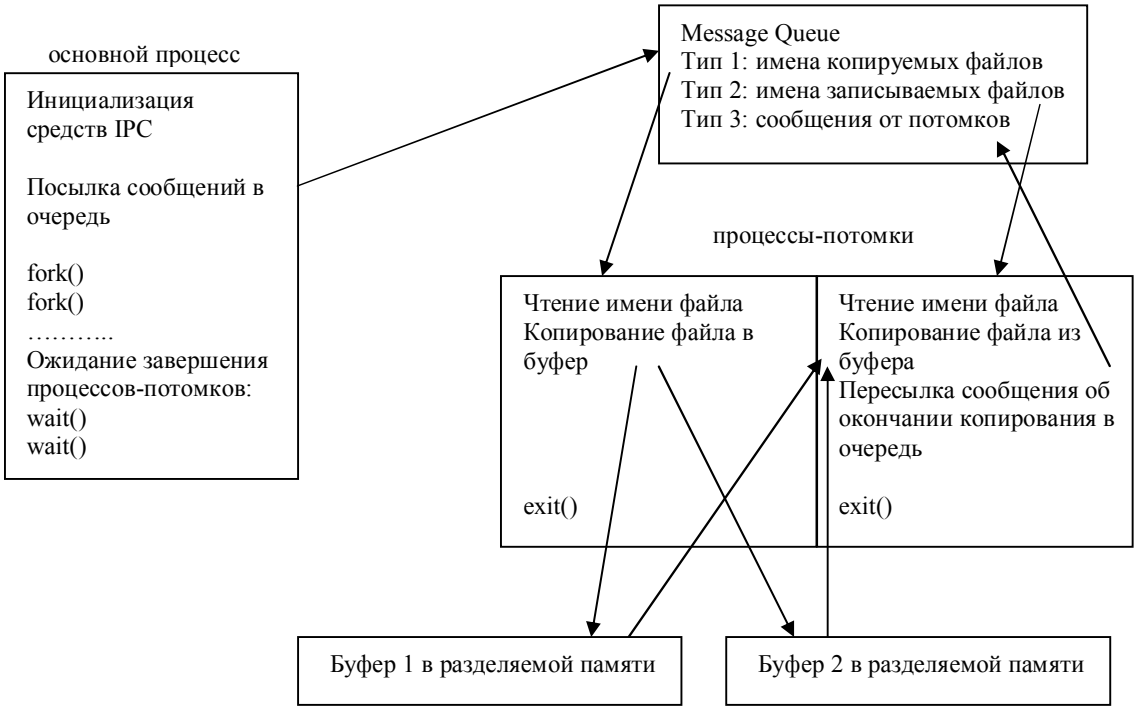
Если же к сегменту были подключены процессы, то в элементе таблицы сегментов выставляется признак, запрещающий дальнейшие подключения к сегменту других процессов. После того, как последний процесс, успевший подключить сегмент, заканчивает работу, происходит физическое удаление сегмента из памяти и как объекта IPC.

Пример взаимодействия процессов посредством механизма IPC:

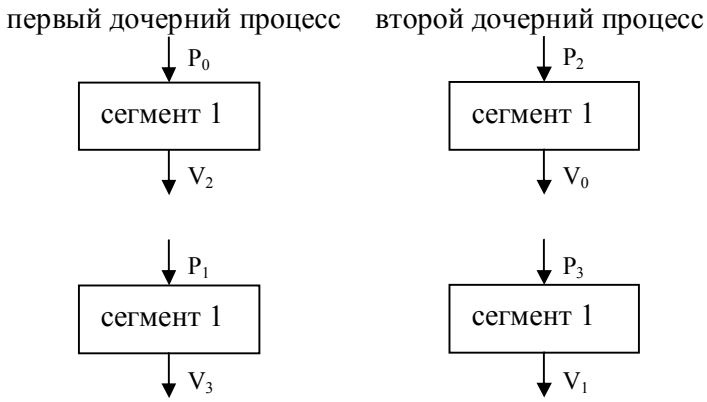
Исходный процесс порождает два процесса-потомка, информацией с которыми обменивается посредством очереди сообщений. Порождённые процессы выполняют копирование больших файлов, выполняя операцию асинхронно с использованием двух буферов в разделяемой памяти. Синхронизация работы с буфером выполняется посредством аппарата семафоров.

Основной процесс посредством очереди сообщений передаёт потомку имена копируемых файлов и некоторое ключевое слово quit, сигнализирующее об окончании операции копирования.

Процесс-потомок, выполняющий запись, посредством этой же очереди извещает процесс-потомок, выполняющий чтение, о завершении записи очередного файла.



Синхронизация:



Текст программы:

```
/*   пример использования очередей сообщений, семафоров и разделяемой
     памяти в задаче копирования файлов.
     Параметров нет.
     Имена исходных файлов: source_file1.txt, source_file2.ec
     (см. массив Table table_names)
*/
#include<stdio.h>
#include<stdlib.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<sys/shm.h>
#include<sys/msg.h>
#include<errno.h>

#define ERR ((struct databuf *) -1)

struct databuf                  /* структура с данными и счетчиком чтения */
{
    int d_nread;
    char d_buf[262144];
};

/* следующий тип должен быть определен взависимости от конфигурации ОС */
typedef union semun
{
     int val;
     struct semid_ds *buf;
     ushort_t *array;
} semun;

typedef struct                  /* структура с именами копируемых файлов */
{
     int len1;
     char *name1;
     int len2;
     char *name2;
} Table;

/* массив структур с именами копируемых файлов */
static Table table_names[] = {
     {16, "source_file1.txt", 16, "target_file1.txt"},
     {15, "source_file2.ec", 14, "target_file2.c"},
     {4, "quit", 4, "quit"},
     {-1, NULL, -1, NULL}
};

int initsem (key_t semkey, int num)
/* функция инициализации набора из num семафоров */
{
     int status = 0, semid, i;
     union semun arg;
     if ((semid = semget (semkey, num, 0666 | IPC_CREAT | IPC_EXCL)) == -1)
     {
               if (errno == EEXIST) /* если набор семафоров существовал ранее */
                         semid = semget (semkey, num, 0); /* то открыть его */
     }
```

```
    if (semid == -1)
            return -1;
    return (semid);
}

int initmsg (key_t msgkey) /* функция инициализации очереди */
{
    int msgid;
    if ((msgid = msgget (msgkey, 0600 | IPC_CREAT | IPC_EXCL)) == -1)
    {
            if (errno == EEXIST) /* если очередь существовала ранее */
                    msgid = msgget (msgkey, 0); /* то открыть ее */
    }
    if (msgid == -1)
            return -1;
    return (msgid);
}

int getseg (key_t segkey, struct databuf **p)
/* функция инициализации и присоединения сегмента памяти */
{
    int segid;
    if ((segid = shmget (segkey, sizeof(struct databuf),
                    0600 | IPC_CREAT |IPC_EXCL)) == -1)
    {
            if (errno == EEXIST) /* если сегмент существовал ранее */
                    /* то подключить его */
                    segid = shmget (segkey, sizeof(struct databuf), 0);
    }
    if (segid == -1)
            return -1;
    if ((*p = (struct databuf*) shmat (segid, 0, 0)) == ERR)
            return -1;
    return segid;
}

int p (int semid, int num)
/* выполнение функции p() над семафором num */
{
    struct sembuf p_buf;
    p_buf.sem_num = num;
    p_buf.sem_op = -1;
    p_buf.sem_flg = 0;
    if (semop (semid, &p_buf, 1) == -1)
    {
            printf ("error in p %d\n", semop (semid, &p_buf, 1));
            exit (1);
    }
    return 0;
}

int v (int semid, int num)
/* выполнение функции v() над семафором num */
{
    struct sembuf p_buf;
    p_buf.sem_num = num;
    p_buf.sem_op = 1;
    p_buf.sem_flg = 0;
    if (semop (semid, &p_buf, 1) == -1)
```

```
    {
            printf ("error in v %d\n", semop (semid, &p_buf, 1));
            exit (1);
    }
    return 0;
}

void main()
{
    /* ключи средств IPC */
    key_t semkey = 0X200, msgkey = 0X201, segkey1 = 0X202, segkey2 = 0X203;
    /* размер сегмента разделяемой памяти */
    size_t size = 262144, len;
    int i, num, semid, msgid, segid1, segid2, ret_code, status;
    int fd1;            /* дескриптор */
    pid_t pid;
    char name_file[20], buf[100];
    union semun arg;        /* для инициализации семафора */
    struct msqid_ds msq_stat;
    struct shmid_ds shm_stat;
    struct q_entry              /* структура для работы с очередью */
    {
            long mtype;
            char mtext[20];
    };
    struct databuf *buf1, *buf2;
    struct q_entry s_entry, t_entry;
    Table *tb;                  /* указатель на массив структур */
    /* инициализация семафоров, очередей, разделяемлй памяти */
    if ((semid = initsem (semkey, 4)) < 0)
    {
            printf ("semid= %d   errno= %d\n", semid, errno);
            exit (1);
    }
    if ((msgid = initmsg (msgkey)) < 0)
    {
            printf ("msgid= %d\n", msgid);
            exit (1);
    }
    if ((segid1 = getseg (segkey1, &buf1)) < 0)
            exit (1);
    if ((segid2 = getseg (segkey2, &buf2)) < 0)
            exit (1);
    for (tb = table_names; tb->name1 != NULL; tb++)
    {
            s_entry.mtype = 1;
            len = tb->len1;
            strncpy (s_entry.mtext, tb->name1, 20);
            if (msgsnd (msgid, &s_entry, len, 0) == -1)
            {
                    printf ("error of send message\n");
                    return -1;
            }       /* выбрали информацию из массива и послали в первую очередь */
            s_entry.mtype = 2;
            len = tb->len2;
            strncpy (s_entry.mtext, tb->name2, 20);
            if (msgsnd (msgid, &s_entry, len, 0) == -1)
            {
                    printf ("error of send message\n");
```

```
                return -1;
        }       /* выбрали информацию из массива и послали во вторую очередь */

}
/* инициализировать значения семафоров */
for (i = 0; i < 4; i++)
{
        arg.val = 0;    /* начальное значение семафора */
        status = semctl (semid, i, SETVAL, arg);
}
v (semid,0);            /* значения первого и второго семафоров установить в 1 */
v (semid,1);
if (fork() == 0)
{
        /* первый потомок */
        pid = getpid();
        for (;;)
        {
                if ((status = msgrcv (msgid, &t_entry, 20, 1, 0)) == -1)
                {
                        printf ("error of get message\n");
                        return -1;
                }       /* выбрали информацию из первой очереди */
                if (strncmp (t_entry.mtext, "quit", status) == 0)
                        exit (0);
                strncpy (name_file, t_entry.mtext, status);
                name_file[status] = '\0'; /* выбрали имя очередного файла */
                fd1 = open (name_file, O_RDONLY);
                for(;;)
                {
                        /* читаем в 1-й буфер, если позволяет семафор */
                        p (semid,0);
                        status = read (fd1, buf1->d_buf, 262144);
                        if (status == -1)
                        {
                                printf ("read error\n");
                                exit (1);
                        }
                        buf1->d_nread = status;
                        v (semid,2);
                        if (status == 0)
                                break;
                        /* читаем во 2-й буфер, если позволяет семафор */

                        p (semid,1);
                        status = read (fd1, buf2->d_buf, 262144);
                        if (status == -1)
                        {
                                printf("read error\n");
                                exit (1);
                        }
                        buf2->d_nread = status;
                        v (semid,3);
                        if (status == 0)
                                break;
                }
                close (fd1);
                /* дождаться подтверждения об окончании записи файла */
                if ((status = msgrcv (msgid, &t_entry, 20, 3, 0)) == -1)
```

```
                        {
                                printf ("error of get message\n");
                                return -1;
                        }       /* выбрали информацию из третьей очереди */
                        if (strncmp (t_entry.mtext, "finish", status) == 0)
                        {
                                /* инициализировать значения семафоров для следующей
итерации */
                                for (i = 0; i < 4; i++)
                                {
                                        arg.val = 0; /* начальное значение семафоров */
                                        status = semctl (semid, i, SETVAL, arg);
                                }
                                /* значения первого и второго семафоров установить в 1 */
                                v (semid,0);
                                v (semid,1);
                        }
                }
        }
        else if (fork() == 0)
        {
                /* второй потомок */
                pid = getpid();
                for (;;)
                {
                        if ((status = msgrcv (msgid, &t_entry, 20, 2, 0)) == -1)
                        {
                                printf ("error of get message\n");
                                return -1;
                        }       /* выбрали информацию из второй очереди */
                        if (strncmp (t_entry.mtext, "quit", status) == 0)
                                exit (0);
                        strncpy (name_file, t_entry.mtext, status);
                        name_file[status] = '\0'; /* выбрали имя очередного файла */
                        fd1 = open (name_file, O_CREAT | O_WRONLY, 0666);
                        for(;;)
                        {
                                /* пишем из 1-го буфера, если позволяет семафор */
                                p (semid, 2);
                                if (buf1->d_nread == 0)
                                        break;
                                status = write (fd1, buf1->d_buf, buf1->d_nread);
                                v (semid, 0);
                                if (status == -1)
                                {
                                        printf ("write error\n");
                                        exit (1);
                                }
                                /* пишем из 2-го буфера, если позволяет семафор */
                                p (semid, 3);
                                if (buf2->d_nread == 0) break;
                                status = write (fd1, buf2->d_buf, buf2->d_nread);
                                v (semid, 1);
                                if (status == -1)
                                {
                                        printf ("write error\n");
                                        exit (1);
                                }
                        }
```

```
                        /* инициализировать значения семафоров для следующей итерации */
                        for (i = 0; i < 4; i++)
                        {
                                arg.val = 0; /* начальное значение семафоров */
                                status = semctl (semid, i, SETVAL, arg);
                        }
                        /* послать подтверждения об окончании записи файла */
                        s_entry.mtype = 3;
                        len = 6;
                        strncpy (s_entry.mtext, "finish", 20);
                        if (msgsnd (msgid, &s_entry, len, 0) == -1)
                        {
                                printf ("error of send message\n");
                                return -1;
                        }       /* послали сообщение об окончании записи в 3-ю очередь */
                        close (fd1);
                }
        }
        else
                for (i = 0; i < 2; i++)
                        wait (&ret_code); /* ждем завершения потомков */
        /* удаление средств IPC */
        status = semctl (semid, 0, IPC_RMID, arg);
        if (status == -1)
                printf ("error of RMD sem\n");
        status = msgctl (msgid, IPC_RMID, &msq_stat);
        if (status == -1)
                printf ("error of RMD mes\n");
        status = shmctl (segid1, IPC_RMID, &shm_stat);
        if (status == -1)
                printf ("error of RMD seg1\n");
        status = shmctl (segid2, IPC_RMID, &shm_stat);
        if (status == -1)
                printf ("error of RMD seg2\n");
}
```

Взаимодействие процессов в сетевом варианте.

В данном варианте взаимодействия процессов участвуют серверный и клиентский процессы, сообщающиеся между собой по технологии «клиент-сервер».

Существуют следующие типы серверов:

- почтовый;
- www;
- баз данных;
- файловый;
- сервер приложений – программы, предоставляющие другим программам услуги посредством обладания какого-либо ресурса и обеспечения доступа к этому ресурсу.

Технология «клиент-сервер» предполагает, что начальная и конечная обработка выполняется клиентской машиной или программой, а вся основная работа по предоставлению услуг производится серверной программой.

К технологии «клиент-сервер» относятся все типы серверов, представленные выше, кроме файлового.

Основные элементы взаимодействия процессов по сети:

1. Адресация и создание адреса: физический адрес, символьный адрес, сетевой IP-адрес, классы и структура IP-адресов.

2. Серверный порт – то, посредством чего выполняется общение клиентского и серверного процессов.

Для связи с серверным процессом клиентский процесс должен выполнить запрос на подключение к определённому порту серверной машины.

В UNIX-подобных системах файл /etc/services содержит следующие записи:

<имя сервиса (имя серверной программы)> <номер порта>/<протокол>

В общем случае, порты с номером менее 1024 зарезервированы на сетевом уровне для определённых приложений (telnet, FTP, HTTP, IRC и т.д.).

Для портов с номером, большим 1024, программист сам имеет возможность выбирать тот порт, который будет выполнять функции связи.

3. Две модели взаимодействия между процессами в сети: модель соединений (с протоколом TCP) и модель дейтаграмм (протокол UDP).

Совокупная информация об адресе, порте программы-адресата, модели соединения и протокола составляет т. н. *сокет*.

Существует несколько видов сокетов:

- обобщённый, структура которого определена в файле **<sys/socket.h>**:

```
struct sockaddr
{
    u_char sa_family;       //семейство адресов, домен
    char sa_data[];         //адрес сокета
}
```

-сокет для связи через сети, структура которого определена в файле **<netinet.h>**:

```
struct sockaddr_in
{
    u_char sin_len;//только для FreeBSD, размер сокета
    u_char sin_family;      //семейство адресов, домен
    u_short sin_port;       //номер порта
    struct in_addr sin_addr;
    char sin_zero;
}
```

```
struct in_addr
{
    n_int32_t s_addr; //IP-адрес определённого формата
}
```

Сетевые вызовы ОС UNIX не работают с IP-адресами в традиционной форме (т.е. с адресами вида 217.71.130.136). На программном уровне IP-адрес хранится в структуре in_addr в специальном формате. Для преобразования IP-адреса из традиционной формы в форму структуры in_addr используется системный вызов **inet_addr()**.

```
in_addr_t inet_addr (const char *ip_address);
```

Преобразование можно представить следующим образом:

```
…
sockaddr_in server;
…
server.sin_addr.s_addr = inet_addr ("197.124.10.1");
```

Обратное преобразование выполняется системным вызовом **inet_ntoa()**.

```
char *inet_ntoa(struct in_addr in);
```

Для того чтобы процесс мог ссылаться на адрес своего компьютера, в файле **<netinet/in.h>** определена переменная INADDR_ANY, которая содержит локальный адрес своего компьютера в формате in_addr.

Вне зависимости от используемой модели, и клиент, и сервер должны создать сокеты и получить их дескрипторы, используемые в последующих операциях для работы с сокетами.

Создание сокета проводится системным вызовом **socket()**.

```
int socket (int domain, int type, int protocol);
```

domain может принимать следующие значения:

- AF_INET – связь между процессами осуществляется через сеть;
- AF_UNIX – оба процесса находятся на одном компьютере.

type указывает тип соединения:

- SOCK_STREAM – модель соединений;
- SOCK_DGRAM – модель дейтаграмм.

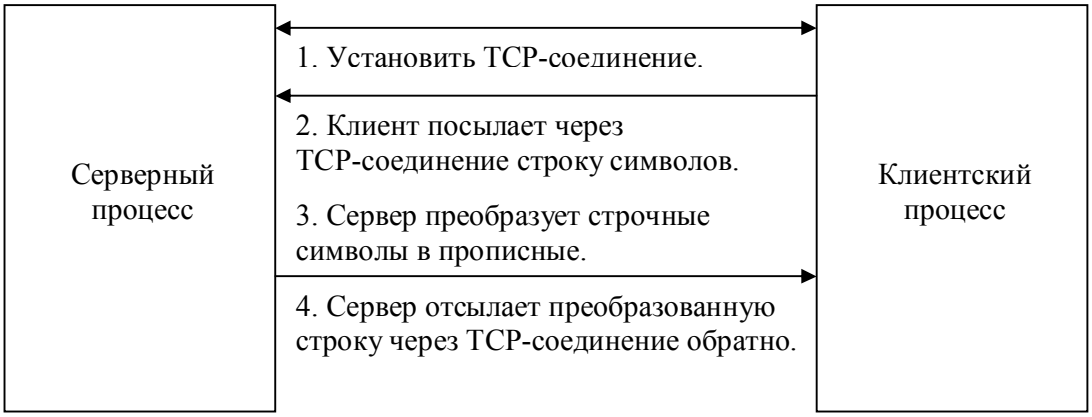
protocol может принимать значения в зависимости от используемого протокола (TCP или UDP), 0 – выбор протокола по умолчанию.

Пример:

Занесение в структуру IP-адреса и создание сокета.

```
…
int sockfd;
struct sockaddr_in server = {0, AF_INET, 7000};
…
/*преобразование и сохранение IP-адреса сервера*/
server.sin_addr.s_addr = inet_addr ("127.0.0.1");
…
/*создание сокета*/
if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) == -1)
{
    /*ошибка при создании сокета*/
}
```

Модель взаимодействия посредством TCP-соединения.
Схема взаимодействия через TCP-соединение.



Структура серверного процесса:

```
<создать сокет (socket())>                                  (1)
<связать адрес сервера с сокетом (bind())>                  (2)
<включить приём TCP-соединений (listen())>                  (3)
for(;;)
{
    <установить соединение с клиентом (accept())>           (4)
    <создать дочерний процесс для работы с соединениями>    (5)
    if (дочерний процесс) /*для всех дочерних процессов*/
    {
            <организовать приём данных от клиента (recv())> (6)
            <преобразовать поступившие данные>
            <отослать ответ клиенту (send())>               (6)
            <закрыть TCP-соединение (close())>              (7)
    }
}
```

Структура клиентского процесса:

```
<преобразовать и записать IP-адрес в структуру сокета (inet_addr())>   (1)
<создать сокет (socket())>                                             (1)
<подключиться к серверу, связав сокет с адресом сервера (connect())>   (5)
for(;;)
{
    <послать сообщение (send())>                                       (6)
    <принять сообщение (recv())>                                       (6)
}
<закрыть TCP-соединение (close())>                                     (7)
```

Отмеченные символом (1) в структурах процессов действия уже разобраны ранее. Рассмотрим остальные системные вызовы:

2. Системный вызов **bind()** – связывание адреса компьютера с сокетом.

```
int bind (int sockfd, const struct sockaddr *address, size_t addr_len);
```

address – указатель на обобщённую структуру сокета;
add_len – размер данной структуры.

Пример:

```
#define SIZE sizeof (struct sockaddr_in)
int sockfd;
struct sockaddr_in server = {0, AF_INET, 7000, INADDR_ANY};
…
/*связать адрес сервера с сокетом*/
if (bind (sockfd, (struct sockaddr*) &server, SIZE) == -1)
{
    /*ошибка вызова bind*/
    exit (1);
}
```

3. Системный вызов **listen()** - включение приёма TCP-соединений.

```
int listen (int sockfd, int queue_size);
```

queue_size – число запросов на соединение, которое может стоять в очереди.

Пример:

```
int sockfd;
…
/*включить приём соединений*/
if (listen (sockfd, 5) == -1)
{
    /*ошибка вызова listen*/
    exit (1);
}
```

4. Приём запроса на установку TCP-соединения.

Когда сервер получает от клиента запрос на установку соединения, он (сервер) создаёт новый сокет для работы с этим соединением. Этот дополнительный сокет для работы с соединением создаётся системным вызовом **accept()**, имеющим следующий вид:

```
int accept (int sockfd, struct sockaddr *address, size_t *addr_len);
```

address – указатель на обобщённую структуру сокета с информацией о клиенте;

addr_len – указатель на размер структуры сокета.

В ряде случаев, когда серверному процессу не обязательно знать информацию об адресе клиента, второй аргумент может иметь значение NULL.

Возвращаемое системным вызовом **accept()** значение является идентификатором нового сокета, который будет использоваться для связи.

Пример:

```
int sockfd, newsockfd;
/*приём запроса на соединение*/
if ((newsockfd = accept (sockfd, NULL, NULL)) == -1)
{
    /*ошибка*/
}
```

Данный вариант используется, если необходимость знания адреса клиента отсутствует. Если же необходимо владеть информацией об адресе клиента, то используется такой вариант:

```
#define SIZE sizeof (struct sockaddr_in)
int sockfd, newsockfd;
struct sockaddr_in client;
int client_len = SIZE;
…
if ((newsockfd = accept (sockfd, (struct sockaddr*) &client, &client_len)) == -1)
{
    /*ошибка*/
}
```

5. Подключение к серверу (выполняется только в клиентском процессе) осуществляется при помощи системного вызова **connect()**, имеющего следующий вид:

```
int connect (int sockfd, const struct sockaddr *address, size_t addr_len);
```

Пример:

```
#define SIZE sizeof (struct sockaddr_in)
int sockfd;
struck sockaddr_in server = {0, AF_INET, 7000};
…
/*подключить сокет к адресу сервера*/
if ((connect (sockfd, (struct sockaddr*) &server, SIZE) == -1)
{
    /*ошибка*/
}
```

Перед использованием системного вызова **connect()** необходимо в структуру сокета сервера добавить его (сервера) IP-адрес.

6. Средства приёма-передачи сообщений – системные вызовы **recv()** и **send()**.

Данные системные вызовы по назначению аналогичны системным вызовам **read()** и **write()**, но, в отличие от системных вызовов **read()** и **write()**, используют дополнительные параметры, управляющие пересылкой данных по сети.

Вместо системных вызовов **recv()** и **send()** можно использовать системные вызовы **read()** и **write()**, но при этом теряются возможности управления пересылкой данных по сети.

7. Закрытие соединения.

Осуществляется системным вызовом **close()** с единственным аргументом – идентификатором сокета. Перед выполнением этого системного вызова необходимо корректно завершить и серверный, и клиентский процессы. В противном случае в сети могут возникнуть коллизии.

Тексты программ клиента и сервера из примера:

Сервер:

```
#include <signal.h>
#include <ctype.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <tcpd.h>
#include <sys/fcntl.h>
#include <unistd.h>

#define SIZE sizeof (struct sockaddr_in)

void catcher (int sig);
int newsockfd;

void main ()
{
	int sockfd;
	char c;
	struct sockaddr_in server = {0, AF_INET, 7000, INADDR_ANY};
	static struct sigaction act;
	struct sockaddr_in client;
	int client_len = SIZE;

	signal (SIGPIPE, catcher);
	/*Установить абонентскую точку сокета*/
	if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) == -1)
	{
		perror ("Error of socket");
		exit (1);
	}

	/*Связать адрес с абонентской точкой*/
	if (bind (sockfd, (struct sockaddr*) &server, SIZE) == -1)
	{
		perror ("Error of bind");
		exit (1);
	}

	/*Включить прием соединений*/
	if (listen (sockfd, 5) == -1)
	{
		perror ("Error of listen");
		exit (1);
	}

	for (;;)
	{
		/*Прием запроса на соединение*/
		if ((newsockfd = accept (sockfd, (struct sockaddr*) &client, &client_len)) == -1)
		{
			perror ("Error of accept");
			continue;
		}
		/*Принят запрос на соединение*/
		printf ("request from IP: %s\n", inet_ntoa (client.sin_addr));
```

```
                /*Создать дочерний процесс для работы с этим соединением*/
                if (fork () == 0)
                {
                        while (recv (newsockfd, &c, 1, 0) > 0)
                        {
                                c = toupper (c);
                                send (newsockfd, &c, 1, 0);
                        }
                        /*После того, как клиент прекратит передачу данных,
                        * сокет может быть закрыт и дочерний процесс
                        * завершает работу*/
                        close (newsockfd);
                        exit (0);
                }

                /*В родительском процессе newsockfd не нужен*/
                close (newsockfd);
        }
}

void catcher (int sig)
{
        close (newsockfd);
        exit (0);
}
```

Клиент:

```
#include <ctype.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <tcpd.h>
#include <sys/fcntl.h>
#include <unistd.h>

#define SIZE sizeof (struct sockaddr_in)
void main ()
{
        int sockfd;
        char c, rc;
        int i;
        char mas[5] = "asdfg";
        struct sockaddr_in server = {0, AF_INET, 7000};

        /*Преобразовать и сохранить IP address сервера*/
        server.sin_addr.s_addr = inet_addr ("193.125.2.130");

        /*Установить абонентскую точку сокета*/
        if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) == -1)
        {
                perror ("Error of socket");
                exit (1);
        }

        /*Подключить сокет к адресу сервера*/
        if (connect (sockfd, (struct sockaddr*) &server, SIZE) == -1)
        {
```

```
                perror ("Error of connect");
                exit (1);
        }

        /*Обмен данными с сервером*/
        for (rc = '\n';;)
        {
                if (rc == '\n')
                        printf ("Input lower case letter\n");
                c = getchar ();
                send (sockfd, &c, 1, 0);
                if (recv (sockfd, &rc, 1, 0) > 0)
                        printf ("%c", rc);
                else
                {
                        printf ("Server don't answer\n");
                        close (sockfd);
                        exit (1);
                }
        }
}
```

Особенности программирования в режиме дейтаграмм

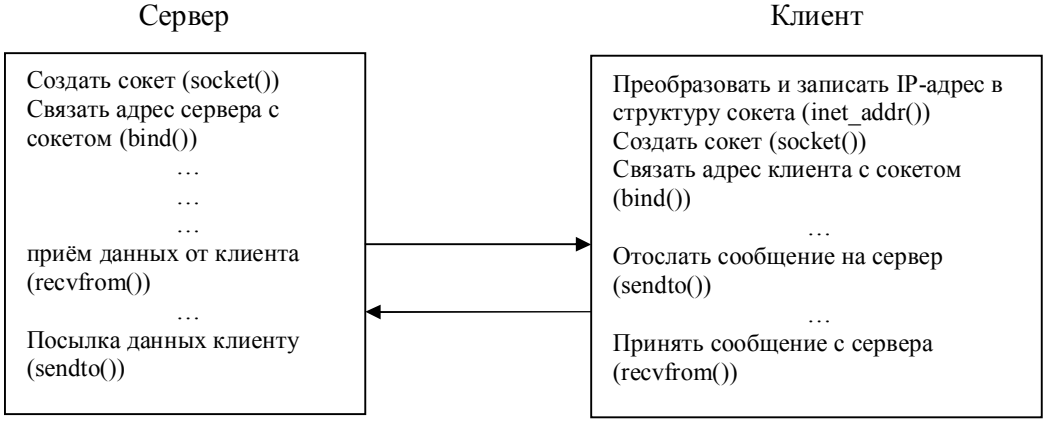
После создания в программе сервера сокета и связывания с ним своего локального адреса, в модели дейтаграмм отсутствуют следующие шаги:

- включение приёма соединений (системный вызов **listen()**);
- запрос на установку соединения (системный вызов **accept()**).

Клиент в режиме дейтаграмм также создаёт сокет, после чего, вместо подключения к серверу, связывает с сокетом свой локальный адрес системным вызовом **bind()**.

В режиме дейтаграмм используются свои системные вызовы для приёма/передачи сообщений: **recvfrom()**, **sendto()**.

Модель взаимодействия в режиме дейтаграмм можно представить следующим образом:



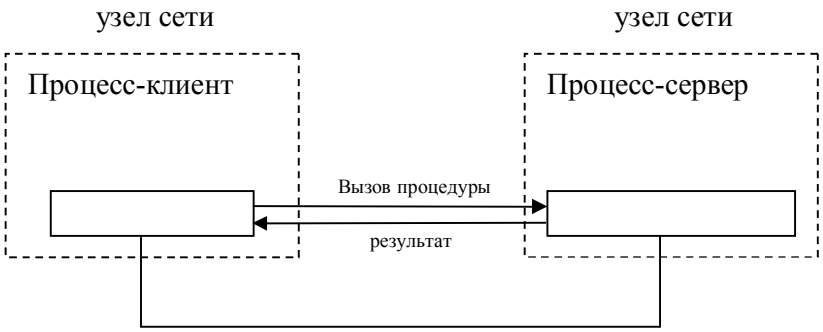
Удалённый вызов процедур (RPC).

Категории Middleware (программное обеспечение промежуточного слоя) – специальный уровень прикладной системы, который расположен между бизнес-приложениями и коммутационным уровнем и изолирует приложения от протоколов и деталей операционной системы.

В эту категорию входят:

1. Программное обеспечение доступа к базам данных.
2. Системы, реализующие RPC.
3. Мониторы транзакций (TP-мониторы).
4. Средства интеграции распределённых объектов.
5. ПО, ориентированное на обработку данных (ПО MOM).
6. Сервера приложений.

Идея RPC – расширение механизма вызова процедур, состоящее в передаче управления и параметров на одной машине на сетевом уровне.



Явное сетевое программирование – программирование вызова удалённых процедур посредством механизма сокетов.

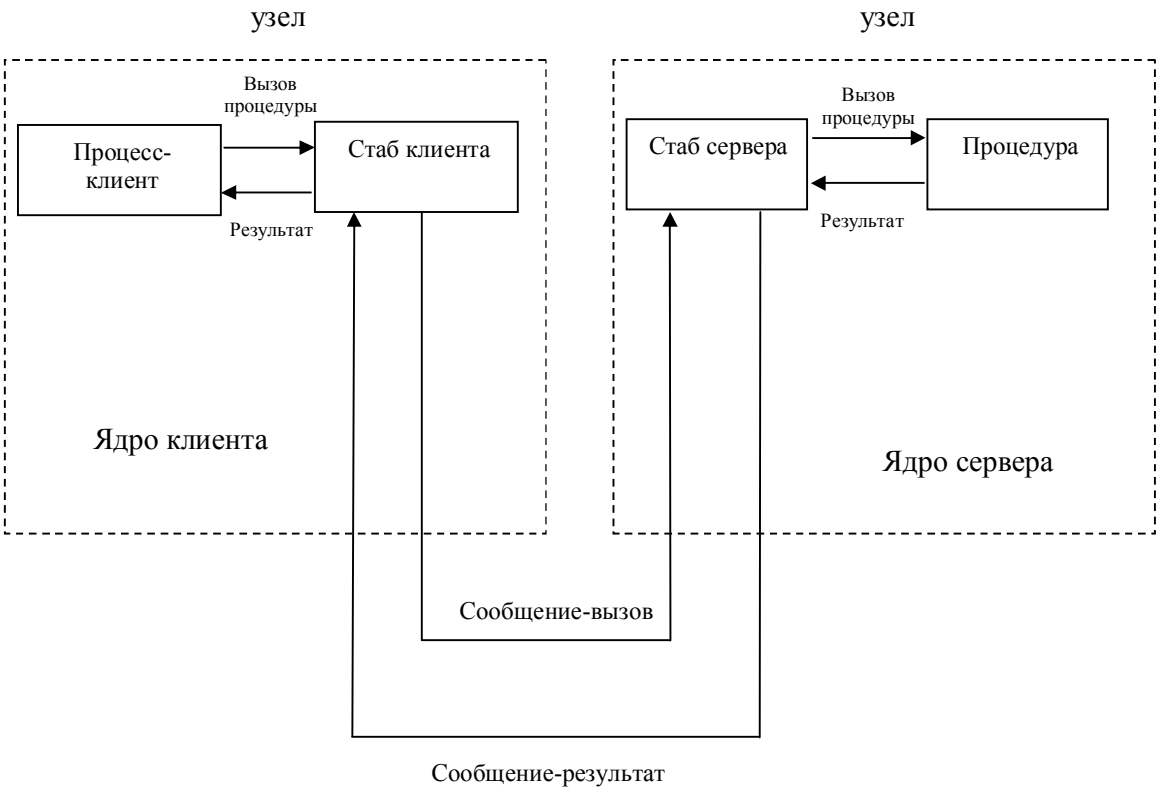
Категории Middleware и механизм RPC являются более высоким уровнем, чем связь посредством сокетов.

Неявное сетевое программирование – программирование с использованием механизма RPC.

Характерными чертами RPC являются:

1. Асимметричность – один из участников взаимодействия является его инициатором.
2. Синхронность – выполнение вызывающей процедуры приостанавливается с момента выдачи запроса и возобновляется только после возврата из вызываемой процедуры.

Идеология механизма RPC: для установки связи, передачи вызова и возврата результата клиентский и серверный процессы обращаются к специальным компонентам, которые носят название клиентского и серверного суррогата (заглушки сервера и клиента, стабы клиента и сервера (client stub, server stub)). Эти стаб-процедуры предназначены для организации взаимодействия прикладных модулей. При вызове удалённой процедуры вызов, вместе с параметрами, передаётся клиентскому стабу, который упаковывает эти данные в сетевое сообщение (эта операция носит название marshalling) и передаёт сообщение серверному стабу. Серверный стаб распаковывает входные данные (unmarshalling), вызывает требуемую процедуру и после её завершения выполняет обратную процедуру упаковки и отсылки данных.



Реализацию механизма RPC на программном уровне можно проиллюстрировать следующим примером: клиент вызывает процедуру сервера с аргументом типа long, возвращаемое значение представляет собой квадрат аргумента.

Этапы:

1. Подготовка файла-спецификации RPC. В данном случае его вид таков:

файл носит название **//sunrpc/square1/square.x** (расширение .x означает, что данный файл является спецификацией RPC).

```
struct square_in
{
    long argl; /*входной аргумент*/
}

struct square_out
{
    long resl; /*результат*/
}

program square_prog
{
    version square_vers
    {
            square_out squareproc (square_in) = 1; /*номер процедуры*/
    } = 1; /*номер версии*/
} = 0x31320000; /*номер программы*/
```

2. Компиляция спецификации некоторой программой, носящей название rpcgen. Командная строка для нашей задачи выглядит следующим образом:

>rpcgen –C square.x

На выходе получается:

- заголовочный файл **square.h** со всеми необходимыми прототипами;
- заглушки для клиента и сервера: **square_clnt.c** и **square_svc.c**;
- программа **square_xdr.c**, выполняющая преобразование данных на стороне клиента и сервера.

3. Подготовка головной функции клиента, которая осуществляет удалённый вызов процедуры.

Исходя из нашей задачи, эта процедура будет выглядеть следующим образом (файл будет носить название **client.c**):

```
#include … /*всё, что нужно*/
#include “square.h” /*создаёт rpcgen*/

/*программа клиента*/
int main (int agrc, char **argv)
{
    CLIENT *cl; /*дескриптор клиента*/
    square_in in;
    square_out *outp;
    if (argc != 3)
            err_quit (“error:client <hostname> <integer_value>”);
    cl clnt_create (argv[1], square_proc, square_vers, "tcp");
    in.argl = atoll (argv[2]);
    if ((out = squareproc_1 (&in.cl)) == NULL) /*вызов удалённой процедуры*/
            printf (“Result: %d\n”, outp->resl;);
    exit (0);
}
```

clnt_create() – функция, создающая клиента RPC, возвращает некоторый ненулевой дескриптор, используемый далее для вызова функции. При этом предполагается, что:

argv[1] – имя или IP-адрес компьютера, содержащего процедуру;
square_proc – имя процедуры;
square_vers – номер версии;
“tcp” – используемый протокол;
argv[2] –содержит число, возводимое в квадрат.

В squareproc_1 используются следующие аргументы:
&in – адрес входной структуры;
cl – созданный клиент RPC.

После имени процедуры вводится суффикс “_1” для указания номера используемой версии программы.

4. Трансляция и сборка головной функции клиента (**client.c**) совместно с файлом **square_clnt.c** (стаб клиента), файлом **square_xdr.c** (функция, осуществляющая преобразование) и библиотекой RPC.

5. Подготовка процедуры сервера, удалённо вызываемой клиентом.
Файл называется **server.c**

```
#include … /*всё, что нужно*/
#include "square.h"

/*процедура сервера*/
square_out* squareproc_1_svc (square_in *inp, struct svc_reg *rpctp)
{
    static square_out out;
    out.resl = inp->argl * inp->argl;
    return (&out);
}
```

Суффикс "_svc" вводится для указания того, что данная функция является серверной.
Первым параметром функции squareproc_1_svc является указатель на входную структуру, второй – структура библиотеки RPC.

6. Трансляция и сборка процедуры server.c совместно с файлом **square_svc.c** (стаб сервера) и **square_xdr.c** (функция, выполняющая преобразование) и библиотекой RPC.

Общая схема может быть проиллюстрирована следующим рисунком:

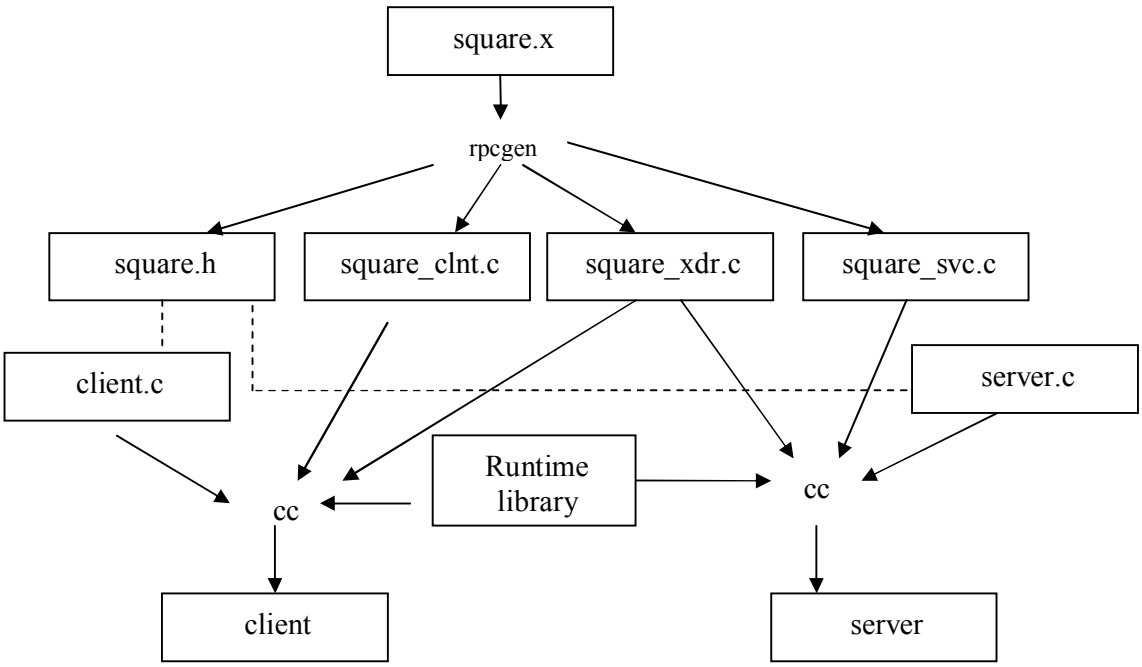
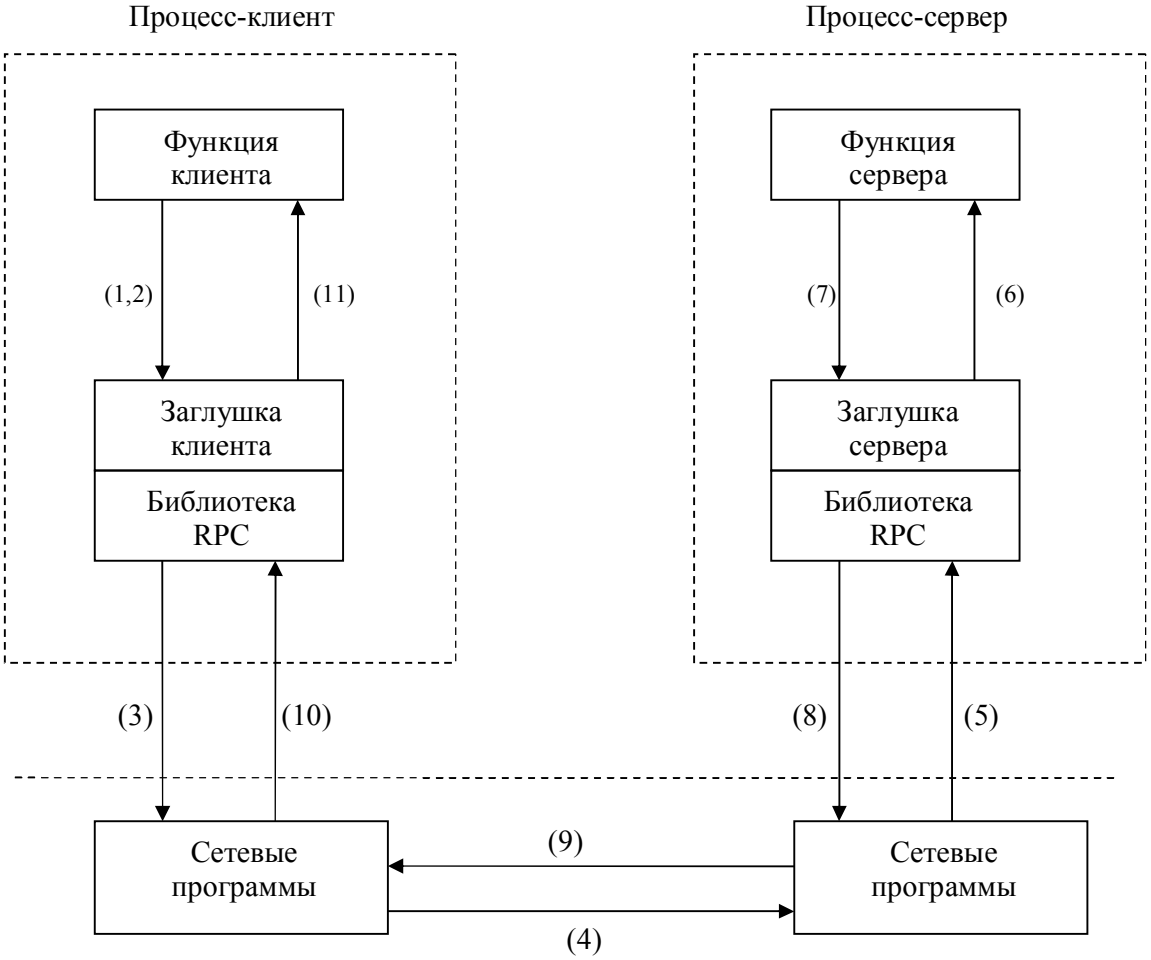


Схема вызова процедуры (порядок действий).



(1) – запуск сервера, который регистрируется в своём узле и управляет работой соответствующих портов; запускается клиент, вызывающий функцию clnt_create() – создание клиента RPC. Эта функция связывается с управляющей программой-сервером, находит нужный порт и устанавливает соединение (в данном случае – по протоколу TCP).

(2) – клиент вызывает локальную процедуру, являющуюся его заглушкой. С точки зрения клиента, именно эта процедура является сервером; заглушка клиента упаковывает аргументы вызова в сетевое сообщение.

(3) – сетевое сообщение отсылается в удалённый узел заглушкой клиента (либо системным вызовом **write()**, либо системным вызовом **sendto()**).

(4) – сетевое сообщение передаётся по протоколу TCP.

(5) – заглушка сервера ожидает запросов от клиента на стороне сервера и, при получении сообщения, распаковывает аргументы и сетевое сообщение.

(6) – заглушка сервера выполняет локальный вызов процедуры для вызова настоящей функции сервера, передавая ей параметры.

(7) – после завершения процедуры сервером управление получает заглушка сервера.

(8) – заглушка сервера упаковывает результаты работы процедуры сервера в сетевое сообщение.

(9) – сообщение передаётся по сети обратно клиенту.

(10) – заглушка клиента считывает сообщение (либо системным вызовом **read**, либо системным вызовом **recvfrom()**).

(11) – после возможного преобразования возвращаемых значений, заглушка клиента передаёт эти значения функции клиента.

Монтирование ФС.

Системный вызов mount() – связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем.

unmount() – исключает файловую систему из этой иерархии.

mount (char *specialfile, char *dir, int rwflag);

specialfile – имя специального файла устройства (на устройстве предполагается наличие ФС);

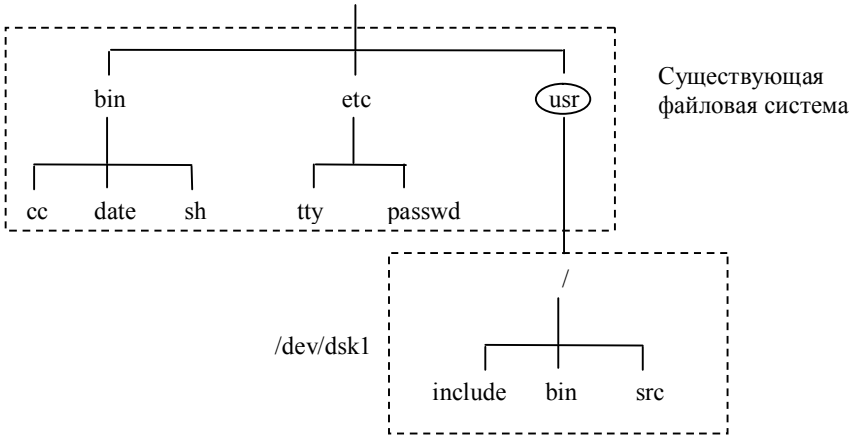
dir – каталог в существующей файловой системе (точка монтирования);

rwflag – набор флагов (например, можно указать, что ФС монтируется только для чтения).

Для выполнения системного вызова **mount()** требуются права суперпользователя.

Пример:

mount ("/dev/dsk1", "/usr", 0);



В ядре существует таблица монтирования файловых систем, каждая запись которой содержит:

- номер устройства, идентифицирующий монтируемую ФС;
- указатель на буфер, где находится суперблок монтируемой ФС;
- указатель на корневой описатель файла монтирования (/ на схеме);
- указатель на описатель файла или каталога, который является точкой монтирования.

Алгоритм mount():

1. Если пользователь не root, возвратить ошибку.
2. Получить описатель файла для блочного специального файла.
3. Проверить допустимость значений параметров.
4. Получить описатель файла для каталога, куда производится монтирование ФС.
5. Если описатель файла не принадлежит каталогу или счётчик ссылок больше единицы:
   - освободить описатель файлов;
   - возвратить ошибку.
6. Найти свободное место в таблице монтирования.
7. Запустить процедуру открытия блочного устройства для данного драйвера.
8. Получить свободный буфер из буферного кэша.
9. Считать суперблок в свободный буфер.
10. Проинициализировать поля суперблока.
11. Получить корневой описатель файла монтируемой системы, сохранив его в таблице монтирования.
12. Сделать пометку, что описатель файла (каталога), являющегося вторым аргументом, является точкой монтирования.

13. Снять блокировку с описателя файла (каталога), который является точкой монтирования.

Системный вызов unmount(), используемый для демонтирования ФС:

unmount (char* specialfile);

При демонтировании ФС ядро обращается к описателю файла демонтируемого устройства, восстанавливает номер устройства для специального файла, освобождает описатель файла и находит в таблице монтирования запись с номером устройства, совпадающим с номером устройства для специального файла (аргумент системного вызова unmount()).

Ядро просматривает таблицу описателей файлов в поисках всех файлов, чей номер устройства совпадает с номером демонтируемой файловой системы. Если файл в текущий момент активен (ему соответствует положительное значение счётчика ссылок), **unmount()** завершается аварийно.

Ядро освобождает корневой описатель файла монтируемой ФС, после этого ядро просматривает буферы в буферном кэше и освобождает те из них, в которых находятся блоки данных, относящиеся к файлам демонтируемой файловой системы.

Ядро в описателе файла, соответствующего точке монтирования, сбрасывает признак, что данный файл является точкой монтирования, освобождает описатель файла, удаляет запись в таблице монтирования.

Связывание.

Системный вызов **link()** - осуществляет связывание файла с ещё одним именем.

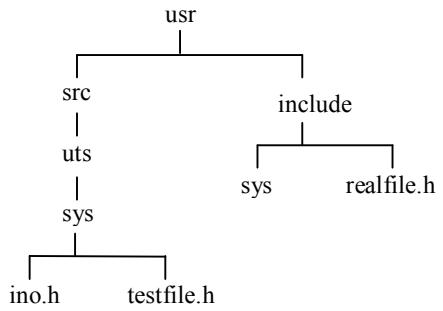
link (char* source_file_name, char *target_file_name);

Связывание позволяет обращаться к одним и тем же файлам или каталогам по нескольким именам.

Пример:

link ("/usr/src/uts/sys", "/usr/include/sys");

link ("/usr/include/realfile.h", "/usr/src/uts/sys/testfile.h");



В результате выполнения этих системных вызовов к одному и тому же файлу можно обращаться, используя разные имена:

1. /usr/src/uts/sys/testfile.h
2. /usr/include/sys/testfile.h
3. /usr/include/realfile.h

Производить связывание для каталогов может только пользователь с правами суперпользователя.

Алгоритм link():

1. Получить описатель файла по существующему имени файла. **(** - алгоритм *namei*)**
2. Если у файла слишком много связей или производится связывание каталога без прав суперпользователя:

- освободить описатель файла;
- возвратить код ошибки.

3. Увеличить значение счётчика связей в описателе файла.
4. Откорректировать дисковую копию описателя файла.
5. Снять блокировку с описателя файла. **(*)**
6. Получить описатель файла родительского каталога для включения нового имени файла.
7. Если файл с новым именем уже существует или существующий и новый файлы находятся в разных ФС:
   - отменить корректировки, сделанные выше;
   - возвратить код ошибки.
8. Создать запись в родительском каталоге для файла с новым именем.
9. Включить в неё новое имя и номер описателя существующего файла.
10. Освободить описатель файла родительского каталога.
11. Освободить описатель существующего файла.

Если в **(*)** не снимать блокировку, то может возникнуть тупиковая ситуация.
Рассмотрим более подробно алгоритм *namei* (на входе – имя пути поиска):

1. Если путь поиска берёт начало от корня:
   рабочий индекс = описатель файла корня.
2. Иначе:
   рабочий индекс = описатель файла текущего каталога
3. Пока путь поиска не закончился:
   а) считать следующую компоненту пути поиска;
   б) проверить соответствие рабочего индекса каталога и прав доступа;
   в) считать каталог;
   г) если компонента соответствует записи в каталоге:
      - получить номер описателя файла для совпавшей компоненты;
      - освободить рабочий индекс;
      - рабочий индекс = описатель файла совпавшей компоненты.
   д) иначе:
      возвратить ошибку об отсутствии описателя файла.

Рассмотрим, что может произойти, если блокировка (*) не была снята.
I. Процесс А: link ("/a/b/c/d", "e/f/g");
Процесс Б: link ("/e/f", "/a/b/c/d/e");

Процесс А	Процесс Б
<u>Разбираем строку /a/b/c/d:</u>	<u>Разбираем строку /e/f/g:</u>
получаем описатель файла a	получаем описатель файла e
освобождаем описатель файла a	освобождаем описатель файла e
…	получаем описатель файла f
получаем описатель файла d	<блокировка не снята>
<блокировка не снята>	
	<u>Разбираем строку /a/b/c/d:</u>
<u>Разбираем строку /e/f/g:</u>	получаем описатель файла a
получаем описатель файла e	освобождаем описатель файла a
освобождаем описатель файла e	…
получаем описатель файла f	запрашиваем описатель файла d
Ожидаем снятия блокировки процессом Б	*Ожидаем снятия блокировки процессом А*

Возможна ситуация самоблокировки:

II. link ("/a/b/c", "/a/b/c/d");

при разборе /a/b/c	…
	получили описатель файла для "c"
	<блокировка не снята>

при разборе /a/b/c/d	…
	запрашиваем описатель файла для "c"
	Ожидаем снятия блокировки

При решении этой проблемы, при разблокировке порождается другая проблема. Для её рассмотрения рассмотрим алгоритм системного вызова unlink().

Системный вызов **unlink()** – удаляет из каталога точку входа для файла.

unlink (char *pathname);

Алгоритм unlink():

1. Получить родительский описатель файла для файла с удаляемой связью.
2. Получить описатель файла для файла с удаляемой связью.
3. Если файл является каталогом и пользователь не имеет прав суперпользователя:
   - освободить описатели файлов и возвратить ошибку.
4. Если файл имеет разделяемый текст и текущее значение счётчика связей равно 1:
   - удалить запись из таблицы областей.
5. В родительском каталоге обнулить номер описателя файла для удаляемой связи.
6. Освободить описатель файла родительского каталога.
7. Уменьшить число связей файла на 1.
8. Освободить описатель файла.

Системные вызовы **link() и unlink()** *создают возможности для конкуренции процессов при использовании операции* (*)*:*

Процесс А переводит имя файла в его описатель ("/a/b/c/d").

Процесс Б удаляет каталог, входящий в путь поиска ("c").

Процесс А	Процесс Б
Просматриваем каталог "a" в поисках "b"	…
Просматриваем каталог "b" в поисках "c"	Удаляем связь с именем "c"
Обнаруживаем, что описатель файла "c" заблокирован	(блокируем описатель файла "c")
	Удаляем связь с именем "c"
Ожидаем снятия блокировки	Прежний описатель файла освобождается, если число связей равно нулю
…	
Вновь пытаемся обратиться к несуществующему описателю файла: число связей равно нулю	
Ошибка	

Пусть процесс С в момент t* создаёт новый каталог и случайно получает тот описатель файла, который был ранее у "c". Данный описатель файла назначается новому каталогу "n". Процесс С снимает блокировку с "n".

Прежний описатель файла "c" (теперь "n") свободен. Процесс А получает описатель файла "n", просматривает каталог "n" в поисках файла "d", что может привести к неожиданным результатам.

Процесс может удалять связь файлов в то время, когда другой или тот же процесс уже открыл данный файл. Поскольку ядро снимает с описателя файла блокировку, по

окончании выполнения операции **open**(), операция **unlink**() завершится успешно. Никакой другой процесс уже не сможет обратиться к файлу по имени удалённой связи, но ранее открывший файл процесс может работать с файлом по его дескриптору.

Пример:

```
void main (int argc, char* argv[])
{
    int fd;
    char buf[1024];
    struct stat statbuf;
    fd = open (argv[1], O_RDONLY);
    if (fd == -1)
            exit();
    if (unlink (argv[1]) == -1) /*удалить связь с открытым файлом*/
            exit();
    if (stat (argv[1], &statbuf) == -1) /*узнать состояние файла по имени*/
            printf ("stat %s завершился неудачно\n", argv[1]);
                    /*как и следовало бы*/
    else
            printf ("stat %s завершился успешно\n", argv[1]);
    if (fstat (fd, &statbuf) == -1) /*узнать состояние файла по его дескриптору*/
            printf ("fstat %s завершился неудачно\n", argv[1]);
    else
            printf ("fstat %s завершился успешно\n", argv[1]); /*как и следовало бы*/
    while (read (fd, buf, sizeof (buf)) > 0)
            printf ("%s", buf);
}
```

Механизмы управления виртуальной памятью

Можно говорить о трех алгоритмах организации памяти:

- страничная организация;
- сегментная организация;
- сегментно-страничная организация.

В случае страничной виртуальной памяти виртуальная память каждого процесса и физическая основная память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальный адрес рассматривается как структура, состоящая из двух полей: первое поле задает номер страницы виртуальной памяти, второе поле задает смещение внутри страницы до адресуемого элемента памяти (в большинстве случаев - байта).

При сегментной организации виртуальный адрес по-прежнему состоит из двух полей - номера сегмента и смещения внутри сегмента. Отличие от страничной организации состоит в том, что сегменты виртуальной памяти могут быть разного размера. В элементе таблицы сегментов помимо физического адреса начала сегмента (если виртуальный сегмент содержится в основной памяти) содержится длина сегмента.

При сегментно-страничной организации виртуальной памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае виртуальный адрес состоит из трех полей: номера сегмента виртуальной памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения - таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

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

Далее будем предполагать только страничную организацию.

Как же достигается возможность наличия виртуальной памяти с размером, существенно превышающим размер оперативной памяти?

В элементе таблицы страниц может быть установлен специальный флаг отсутствия страницы, наличие которого заставляет аппаратуру вместо нормального отображения виртуального адреса в физический прервать выполнение команды и передать управление компоненту операционной системы, выполняющей функции листания по требованию (*"demand paging"*). Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, операционная система удовлетворяет это требование путем выделения страницы основной памяти, перемещения в нее копии страницы, находящейся во внешней памяти, и соответствующей модификации элемента таблицы страниц.

Т.к. размер каждой виртуальной памяти может существенно превосходить размер основной памяти, при выделении страницы основной памяти с большой вероятностью не удастся найти свободную страницу, не приписанную к какой-либо виртуальной памяти. В этом случае операционная система должна в соответствии с заложенными в нее алгоритмами подкачки найти некоторую занятую страницу основной памяти, переместить в случае надобности ее содержимое во внешнюю память, модифицировать соответствующий элемент соответствующей таблицы страниц и после этого продолжить процесс удовлетворения доступа к странице.

Существует большое количество разнообразных алгоритмов подкачки.

Во-первых, алгоритмы подкачки делятся на глобальные и локальные. При использовании глобальных алгоритмов операционная система при потребности замещения ищет страницу основной памяти среди всех страниц, независимо от их принадлежности к какой-либо виртуальной памяти. Локальные алгоритмы предполагают, что если возникает требование доступа к отсутствующей в основной памяти странице виртуальной памяти ВП1, то страница для замещения будет искаться только среди страниц основной памяти, приписанных к той же виртуальной памяти ВП1.

Наиболее распространенными традиционными алгоритмами (как в глобальном, так в локальном вариантах) являются алгоритмы *FIFO (First In First Out)* и *LRU (Least Recently Used)*. При использовании алгоритма FIFO для замещения выбирается страница, которая дольше всего остается приписанной к виртуальной памяти. Алгоритм LRU предполагает, что замещать следует ту страницу, к которой дольше всего не происходили обращения. Хотя интуитивно кажется, что критерий алгоритма LRU является более правильным, известны ситуации, в которых алгоритм FIFO работает лучше (и, кроме того, он гораздо более дешево реализуется).

Замечание. При использовании глобальных алгоритмов, вне зависимости от конкретного применяемого алгоритма, возможны и теоретически неизбежны критические ситуации, которые называются по-английски *thrashing* (не очень точно это определяет русский термин "пробуксовка"). Рассмотрим простой пример.

1. Пусть в мультипрограммном режиме выполняются два процесса – П1 в виртуальной памяти ВП1 и П2 в виртуальной памяти ВП2, причем суммарный размер ВП1 + ВП2 >ОП.

2. Предположим, что в момент времени t1 в процессе П1 возникает требование виртуальной страницы ВС1. ОС обрабатывает прерывание и выбирает для замещения страницу основной памяти С2, приписанную к виртуальной странице ВС2 виртуальной памяти ВП2. Для полной обработки требования доступа к ВС1 в общем случае потребуется два обмена с внешней памятью: первый, чтобы записать текущее содержимое С2, второй - чтобы прочитать копию ВС1.

3. Поскольку ОС поддерживает мультипрограммный режим работы, то во время выполнения обменов доступ к процессору получит процесс П2, и он, вполне вероятно, может потребовать доступа к своей виртуальной странице ВС2, которую у него только что отняли. Опять будет обрабатываться прерывание, и ОС может заменить некоторую страницу основной памяти С3, которая приписана к виртуальной странице ВС3 в ВП1.

4. Когда закончатся обмены, связанные с обработкой требования доступа к ВС1, возобновится процесс П1, и он, вполне вероятно, потребует доступа к своей виртуальной странице ВС3 (которую у него только что отобрали). И так далее. Общий эффект состоит в том, что непрерывно работает ОС, выполняя бесчисленные и бессмысленные обмены с внешней памятью, а пользовательские процессы П1 и П2 практически не продвигаются.

Единственным алгоритмом, теоретически гарантирующим отсутствие *thrashing*, является т.н. *"оптимальный алгоритм Биледи"* (венгерский математик), заключающийся в том, что для замещения следует выбирать страницу, к которой в будущем наиболее долго не будет обращений (т.к. в динамической среде ОС точное знание будущего невозможно, и в этом контексте алгоритм Биледи представляет только теоретический интерес).

Что еще есть?

В свое время (1968 г.) Питер Деннинг сформулировал принцип локальности ссылок (называемый *принципом Деннинга*) и выдвинул идею алгоритма подкачки, основанного на понятии рабочего набора (в некотором смысле предложенный им подход является практически реализуемой аппроксимацией оптимального алгоритма Биледи). Принцип локальности ссылок состоит в том, что если в период времени (T-t, T) программа обращалась к страницам (C1, C2, ..., Cn), то при надлежащем выборе t с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (T, T+t). Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно хорошо его прогнозировать исходя из прошлого. Набор страниц (C1, C2, ..., Cn) называется рабочим набором программы (или, правильнее, соответствующего процесса) в момент времени T. Идея алгоритма подкачки Деннинга состоит в том, что ОС в каждый момент времени должна обеспечивать наличие в основной памяти текущих рабочих наборов всех процессов, которым разрешена конкуренция за доступ к процессору. Во-первых, полная реализация алгоритма Деннинга практически гарантирует отсутствие *thrashing*. Во-вторых, алгоритм реализуем (известна, по меньшей мере, одна его полная реализация, которая однако потребовала специальной аппаратной поддержки). В-третьих, полная реализация алгоритма Деннинга вызывает очень большие накладные расходы.

Поэтому на практике применяются облегченные варианты алгоритмов подкачки, основанных на идее рабочего набора. Рассмотрим один из вариантов, применяемый в ветви System V).

Основная идея заключается в оценке рабочего набора процесса на основе использования аппаратно или программно устанавливаемых признаков обращения к страницам основной памяти.

Периодически для каждого процесса производятся следующие действия. Просматриваются таблицы отображения всех сегментов виртуальной памяти этого процесса. Если элемент таблицы отображения содержит ссылку на описатель физической страницы, то анализируется признак обращения. Если признак установлен, то страница считается входящей в рабочий набор данного процесса, и сбрасывается в нуль счетчик старения данной страницы. Если признак не установлен, то к счетчику старения добавляется единица, а страница приобретает статус кандидата на выход из рабочего набора процесса. Если при этом значение счетчика достигает некоторого критического значения, страница считается вышедшей из рабочего набора процесса, и ее описатель заносится в список страниц, которые можно откачать, если это потребуется, во внешнюю память. По ходу просмотра элементов таблиц отображения в каждом из них признак обращения гасится.

Откачку страниц, не входящих в рабочие наборы процессов, производит специальный системный процесс–stealer. Он начинает работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога. Функцией этого процесса является анализ необходимости откачки страницы на основе признака изменения и запись копии страницы в соответствующую область внешней памяти.

При попытке обращения к виртуальной странице, отсутствующей в основной памяти, возникает аппаратное прерывание к ОС. Если список описателей свободных страниц не пуст, то из него выбирается некоторый описатель, и соответствующая страница подключается к виртуальной памяти процесса.

Если возникает требование страницы в условиях, когда список описателей свободных страниц пуст, то начинает работать механизм своппинга. Основной повод для применения механизма своппинга состоит в том, что простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации *thrashing*, поскольку разрушало бы рабочий набор некоторого процесса. Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится кандидатом на своппинг. Ему больше не предоставляются ресурсы процессора, и описатель процесса ставится в очередь к системному процессу-swapper'у. Конечно, в этой очереди может находиться несколько процессов. Процесс-swapper по очереди осуществляет полный своппинг этих процессов (т.е. откачку всех страниц их виртуальной памяти, которые присутствуют в основной памяти), помещая соответствующие описатели физических страниц в список свободных страниц, до тех пор, пока количество страниц в этом списке не достигнет установленного в системе верхнего предела. После завершения полного своппинга каждого процесса одному из процессов из очереди к процессу–swapper'у дается возможность попытаться продолжить свое выполнение.

В ряде версий ОС используется более упрощенный алгоритм, в котором вероятность thrashing погашается за счет своппинга. Используемый алгоритм называется NRU (Not Recently Used) или clock. Смысл алгоритма состоит в том, что процесс-stealer периодически очищает признаки обращения всех страниц основной памяти, входящих в виртуальную память процессов (отсюда название "clock"). Если возникает потребность в откачке (т.е. достигнут нижний предел размера списка описателей свободных страниц), то stealer выбирает в качестве кандидатов на откачку прежде всего те страницы, к которым не было обращений по записи после последней "очистки" и у которых нет признака модификации (т. е. те, которые можно дешевле освободить). Во вторую очередь выбираются страницы, которые действительно нужно откачивать. Параллельно с этим работает описанный выше алгоритм своппинга, т.е. если возникает требование страницы, а свободных страниц нет, то соответствующий процесс становится кандидатом на своппинг.

Приложение.

Список вопросов, вынесенных на самостоятельное изучение (ОС MS-DOS)

Прерывания.
Аппаратные и программные прерывания.
Назначение регистров CS и IP.
Вектор прерываний.
Структура вектора прерывания.
Таблица векторов прерывания.
Система приоритетов, используемых для управления прерываниями.
Порт.
Блокирование всех прерываний.
Блокирование отдельных прерываний.
Флаг прерывания, регистр флагов.
Инструкции для маскирования прерываний: cli, sti.
Основные прерывания DOS (с 20h по 27h).
Основные функции прерывания 21h.
Функции работы с клавиатурой.
Функции работы с экраном, принтером.
Функции работы с датами и временем.
Функции управления файлами.
Функции управления памятью, разделение памяти.
Прерывания системы BIOS (9, 10, 16; 0÷1Fh).
Средства работы с векторами прерываний (25h, 35h – функции прерывания 21h).
Управление клавиатурой (scan-код).
Назначение адресов 0:0417, 0:0418.
Понятие буфера клавиатуры.
Размер буфера клавиатуры.
ASCII-коды, расширенный код.
Программные средства работы с клавиатурой на уровне BIOS (9h прерывание; адреса 0:0417, 0:0418; 16h прерывание).
Программные средства управления клавиатурой на уровне DOS (функции прерывания 21h).
Основные операции при работе с клавиатурой:
- Очистка буфера клавиатуры на уровне DOS/физическом уровне.
- Проверка наличия символов в буфере клавиатуры - DOS/BIOS/физич. уровень.
- Ожидание ввода символа с клавиатуры - DOS/BIOS.
- Ввод символа с клавиатуры с эхом на экран средствами DOS.
- Получение строки символов средствами DOS.
- Проверка (установка) статуса клавиш-переключателей – BIOS/физич. уровень.

Управление программами:
- 2 формата программ - .exe, .com.
- Содержимое регистров DS, ES при запуске программ.
- Префикс программного сегмента (PSP) и его структура.

Управление оперативной памятью (функции 48h, 49h, 4Ah).
Запуск одной программы из другой программы:
- Блок параметров.
- Назначение и структура блока параметров.
- Понятие строки среды.

Как передать параметры из командной строки (ALT-C, ALT-V…).

Сохранение программы в памяти после её завершения – резидентные программы (2 способа: 27h прерывание, функция 31h прерывания 21h).
Схема установки резидентной программы .exe/.com.
Принципы формирования изображения на экране:
- Образ экрана.
- Соответствие адресов оперативной памяти области экрана.
- Страничный механизм управления памятью.
- Понятие страницы.
- Структура байта-атрибута.
- Средства работы с видеопамятью (BIOS – 10h прерывание, DOS – функции 21h прерывания).
- Операции с видеопамятью в текстовом режиме.
- Сохранение и установка режима и видеостраницы на уровне BIOS.
- Очистка экрана – BIOS/физический уровень.
- Абсолютное позиционирование курсора – BIOS/физический уровень.
- Относительное позиционирование курсора.
- Включение и выключение курсора – BIOS/физический уровень.
- Изменение формы курсора – BIOS/физический уровень.
- Чтение, сохранение и восстановление позиции курсора – BIOS/физический уровень.
- Вывод символа на экран – BIOS/DOS/физический уровень.
- Строка символов на экран – DOS.
- Чтение символа и его атрибутов из данной позиции – BIOS/физический уровень.

Работа с дисковой памятью:
- Логическая модель диска.
- Понятие сектора, кластера – их размеры.
- Структура элемента справочника.
- Понятие FAT-таблицы, её расположение, назначение, размеры.
- Средства работы с отдельными секторами – BIOS (13h прерывание)/DOS (25h, 26h прерывания).
- Работа с FAT-таблицей: преобразование номера кластера в логический номер сектора, нахождение следующего элемента FAT-таблицы.
- Физическая и логическая нумерация диска.

Методы работы с дисковыми файлами:
- Метод управляющего блока (FCB).
- Метод дескрипторов файла.

Определение доступного дискового пространства.
Определение размера файла.
Создание и удаление файлов.
Открытие файла.
Закрытие файла.
Чтение и запись в последовательные файлы.
Чтение из файла и запись в файл.
Основные элементы работы в графическом режиме:
- Регистр палитры.
- Битовые плоскости.
- Управление цветом.
- Изображение точки и линии на экране.

Основы управления внешними устройствами (пример: принтер):
- Регистр входных данных.
- Регистр статуса.
- Регистр управления.