1 Введение

Настоящий документ содержит предварительный проект конспекта лекций по курсу «СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ». Данный конспект может быть использован в качестве дополнительного материала. Окончательный объем курса определяется материалом, реально рассмотренным на лекциях. Данная редакция конспекта может иметь некоторые неоднородности в детализации рассмотрения материала, некоторые из тем рассмотрены более подробно чем на лекциях, некоторые - рассмотрены достаточно поверхностно. Данный дисбаланс авторы постараются исправить в последующих редакциях конспекта. Авторы будут благодарны за получение конструктивных замечаний и обнаруженных в тексте ошибок и неточностей.

2 Вычислительная система. Структура и основные понятия.

Вычислительная система – это совокупность аппаратных и программных средств, функционирующих в единой системе и предназначенных для решения задач определенного класса.

Эксплутационные качества вычислительной системы определяются как свойствами аппаратуры, так и программных компонентов.

2.1 Аппаратные средства ЭВМ

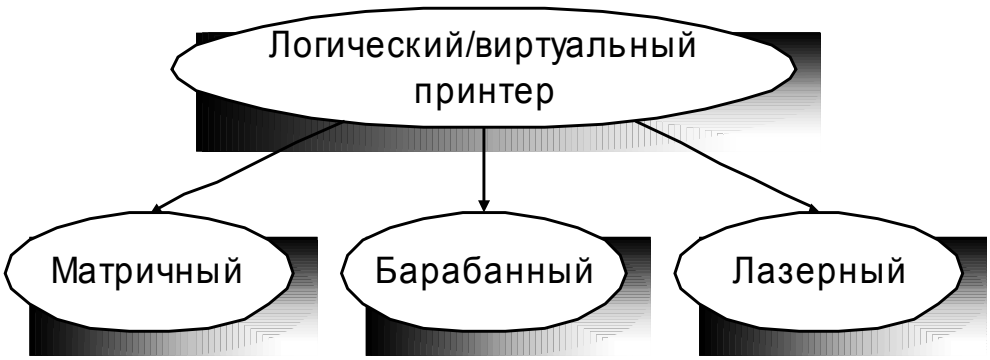
Уровень аппаратных средств ЭВМ – система команд ЭВМ и программно управляемые компоненты ЭВМ.
Программно управляемые компоненты ЭВМ – ***физические ресурсы*** (физические устройства).

2.2 Управление физическими ресурсами

Уровень управления физическими ресурсами – программная составляющая вычислительной системы, обеспечивающая предоставление для каждого конкретного физического ресурса интерфейса для использования – драйвер физического ресурса (устройства).
Драйвер физического устройства – программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством.
Драйвер физического устройства скрывает от пользователя детальные элементы управления конкретным физическим устройством. Драйвер физического устройства ориентирован на конкретные свойства устройства.
На данном уровне иерархии вычислительной системы обеспечивается корректное функционирование и использование физических ресурсов/устройств.

2.3 Управление логическими/виртуальными ресурсами

Виртуальное – нечто реально не существующее, не имеющее реальной физической организации.
Логическое/виртуальное устройство (ресурс) – это устройство/ресурс, некоторые эксплутационные характеристики которого (возможно все) реализованы программным образом.



Иерархия логических/виртуальных устройств (ресурсов).

1-й уровень обобщения

Драйвер логического устройства определенного типа – обобщает интерфейсы драйверов физических устройств этого типа => унификация обращения.

2-й уровень обобщения

Создание логического/виртуального устройства, которому, в конечном счете, соответствует реальное устройство другого типа.

3-й уровень обобщения

Реализация логических/виртуальных устройств (ресурсов) базируется на использовании других логических/виртуальных устройств.

Функция ***управления логическими/виртуальными устройствами*** (ресурсами) – контроль за созданием и использованием.

На уровне управления логическими ресурсами пользователю предоставляется система команд ЭВМ и интерфейсы к драйверам логических/виртуальных устройств/ресурсов.

Уровни управления физическими и логическими устройствами вычислительной системы обычно составляют операционную систему.

2.4 Системы программирования

Уровень ***системы программирования*** обеспечивает поддержание этапов жизни программы: проектирование, кодирование, тестирование, отладка, изготовление программного продукта. На данном уровне пользователю предоставляются средства программирования виртуальной машины, основанные на некотором языке программирования и совокупности доступных логических/виртуальных ресурсов.

2.5 Прикладные системы

Уровень ***прикладных систем*** ориентирован на решение задач из конкретных прикладных областей.

Одной из основных задач настоящего курса является рассмотрение аппаратных и программных компонентов ВС в их взаимосвязи, в функционировании в единой системе.

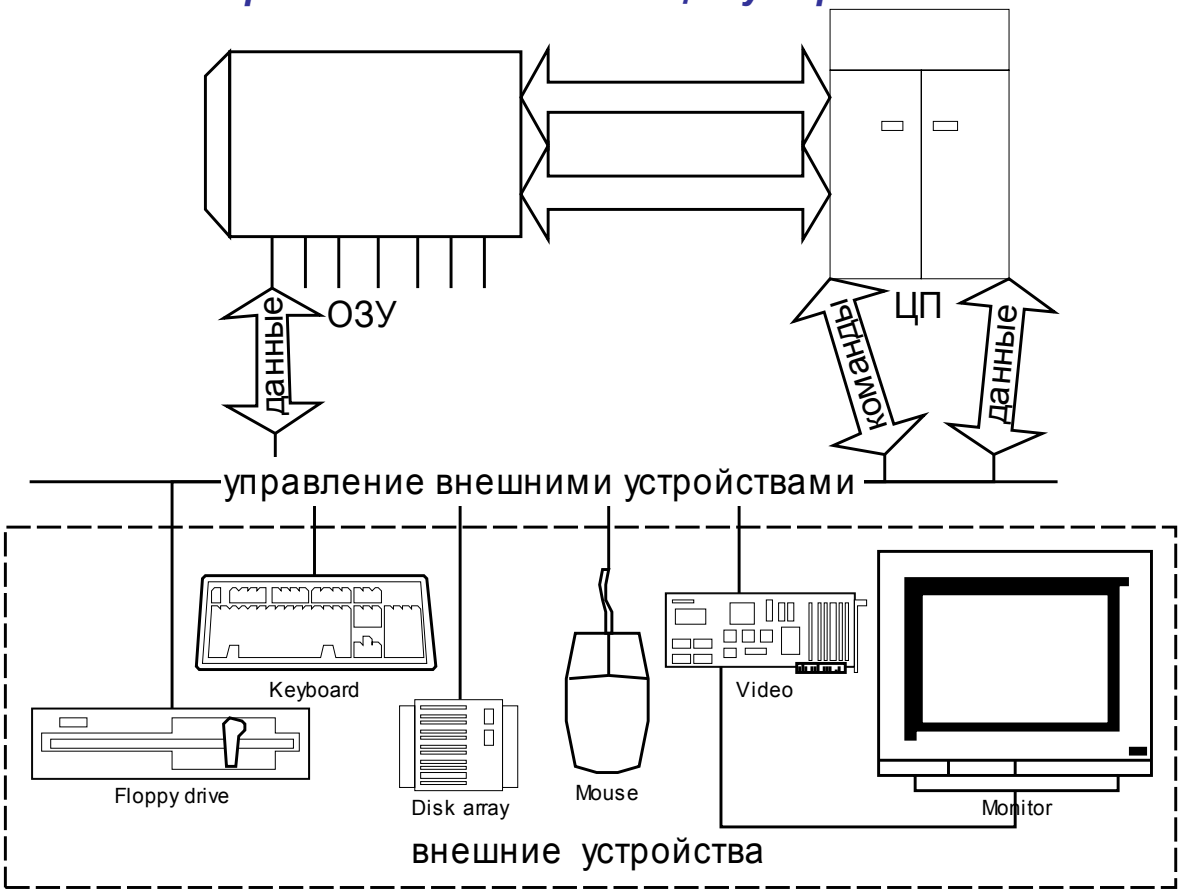
Основной проблемой, возникающей как в ЭВМ в отдельности, так и в вычислительной системе в целом является несоответствие производительности основных компонентов друг другу. Так скорость обработки информации ЦП существенно превосходит скорость доступа к ОЗУ. В свою очередь скорость доступа к внешним устройствам существенно ниже этих показателей для ЦП и ОЗУ и т.д. Так как эти компоненты работают в системе, то на первый взгляд итоговая производительность такой системы будет определяться наименее

“скоростным” компонентом (то есть заведомо основное влияние на системную производительность будет оказывать скорость доступа к ОЗУ, так как обращения в ОЗУ при работе ЭВМ происходят постоянно).

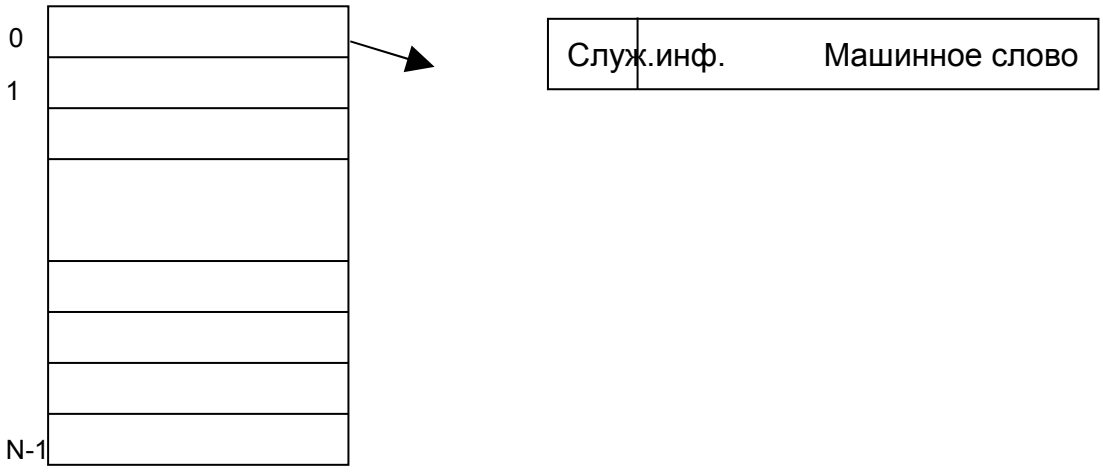
Ниже будут рассмотрены пути решения этой проблемы.

3 Основные компоненты ЭВМ

3.1 Оперативное запоминающее устройство



ОЗУ - устройство, предназначенное для хранения оперативной информации. В ОЗУ размещается исполняемая в данный момент программа и используемые ею данные. ОЗУ состоит из ячеек памяти, содержащей поле машинного слова и поле служебной информации.



Машинное слово – поле программно изменяемой информации, в машинном слове могут располагаться машинные команды (или части машинных команд) или данные, с которыми может оперировать программа. Машинное слово имеет фиксированный для данной ЭВМ размер (обычно размер машинного слова – это количество двоичных разрядов, размещаемых в машинном слове).

Служебная информация (иногда ТЭГ) – поле ячейки памяти, в котором схемами контроля процессора и ОЗУ автоматически размещается информация, необходимая для осуществления контроля за целостностью и корректностью использования данных, размещаемых в машинном слове.

В поле служебной информации могут размещаться:

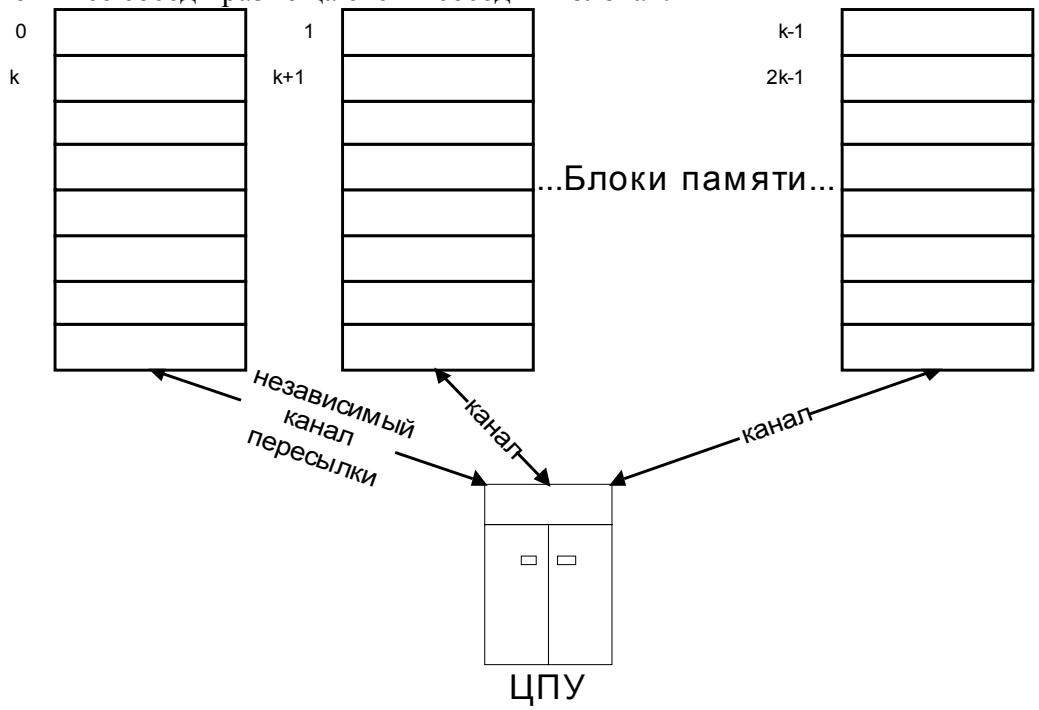
- разряды контроля четности машинного слова (при записи машинного слова подсчет числа единиц в коде машинного слова и дополнение до четного или нечетного в контрольном разряде), при чтении контроль соответствия;
- разряды контроля данные-команда (обеспечение блокировки передачи управления на область данных программы или несанкционированной записи в область команд);
- машинный тип данных – осуществление контроля за соответствием машинной команды и типа ее операндов;

Конкретная структура, а также наличие поля служебной информации зависит от конкретной ЭВМ.

В ОЗУ все ячейки памяти имеют уникальные имена, ***имя - адрес ячейки памяти***. Обычно адрес – это порядковый номер ячейки памяти (нумерация ячеек памяти возможна как подряд идущими номерами, так и номерами, кратными некоторому значению). Доступ к содержимому машинного слова осуществляется посредством использования адреса. Обычно скорость доступа к данным ОЗУ существенно ниже скорости обработки информации в ЦП.

Необходимо, чтобы итоговая скорость выполнения команды процессором как можно меньше зависела от скорости доступа к коду команды и к используемым в ней операндам из памяти. Это составляет проблему, которая системным образом решается на уровне архитектуры ЭВМ.

Расслоение ОЗУ – один из аппаратных путей решения проблемы дисбаланса в скорости доступа к данным, размещенным в ОЗУ и производительностью ЦП. Суть расслоения ОЗУ состоит в следующем. Все ОЗУ состоит из k блоков, каждый из которых может работать независимо. Ячейки памяти распределены между блоками таким образом, что у любой ячейки ее соседи размещаются в соседних блоках.



Возможность предварительной буферизации при чтении команд/данных. Оптимизация при записи в ОЗУ больших объемов данных.

3.2 Центральный процессор

3.2.1 Структура, функции ЦП

ЦП обеспечивает выполнение программы, размещенной в ОЗУ. Осуществляется выбор машинного слова, содержащего очередную машинную команду, дешифрация команды, контроль корректности данных, определение исполнительных адресов операндов, получение значения операндов и исполнение машинной команды.
Регистровая память процессора – сверхоперативные запоминающие устройства, размещенные в процессоре

3.2.2 Регистры общего назначения (РОН)

Используются в машинных командах для организации индексирования и определения исполнительных адресов операндов, а также для хранения значений наиболее часто используемых операндов, в этом случае сокращается число реальных обращений в ОЗУ и повышается системная производительность ЭВМ.

3.2.3 Специальные регистры

Качественный и количественный состав специализированных регистров ЦП зависит от архитектуры ЭВМ. Ниже представлены некоторые из возможных типов регистров, обычно входящие в состав специализированных регистров. Кроме регистров, рассмотренных ниже, мы будем доопределять эту группу по ходу курса.

Регистр адреса (РА) - содержит адрес команды, которая исполняется в данный момент времени. По содержимому РА ЦП осуществляет выборку текущей команды, по завершении ее исполнения регистр адреса изменяет свое значение тем самым указывает на следующую команду, которую необходимо выполнить.

Регистр результата (РР) - содержит код, характеризующий результат выполнения последней арифметико-логической команды. Содержимое РР может характеризовать результат операции. Для арифметических команд это может быть «=0», «>0», «<0», переполнение. Содержимое РР используется для организации ветвлений в программах, а также для программного контроля результатов.

Слово – состояние процессора (ССП или PSW) - регистр, содержащий текущие «настройки» работы процессора и его состояние. Содержание и наличие этого регистра зависит от архитектуры ЭВМ. Например, в ССП может включаться информация о режимах обработки прерываний, режимах выполнения арифметических команд и т. п. Частично, содержимое ССП может устанавливаться специальными командами процессора.

Регистры внешних устройств (РВУ) - специализированные регистры, служащие для организации взаимодействия ЦП с внешними устройствами. Через РВУ осуществляется обмен данными с ВУ и передача управляющей информации (команды управления ВУ и получения кодов результат обработки запросов к ВУ).

Регистр указатель стека - используется для ЭВМ, имеющих аппаратную реализацию стека, в данном регистре размещается адрес вершины стека. Содержимое изменяется автоматически при выполнении «стековых» команд ЦП.

3.3 Буферная память, КЭШ

Вернемся к проблеме дисбаланса скорости доступа к ОЗУ и скорости обработки информации ЦП.

Первое решение – использовать программные средства. Программист может разместить наиболее часто используемые операнды в РОН, тем самым сокращается количество «медленных» обращений в ОЗУ. Результат решения во многом зависит от качества программирования.

Второе решение – использование в архитектуре ЭВМ специальных ***регистровых буферов или КЭШ памяти***.
Регистровые буфера или КЭШ память предназначены для разрешения проблемы несоответствия скоростей работы ОЗУ и ЦП, на аппаратном уровне, т.е. эта форма оптимизации в системе организована аппаратно и работает всегда, вне зависимости от исполняемой программы. Следует отметить, что результат этой оптимизации, в общем случае зависит от характеристик программы (об этом несколько позднее). Традиционно, в развитых ЭВМ используется аппаратная буферизация доступа к операндам команд, а также к самим командам.

3.3.1 Буферизация работы с операндами

Буфер операндов – аппаратная таблица, логически являющаяся компонентом ЦП (физически это может быть и отдельное от ЦП устройство), призванная аппаратно минимизировать количество обращений к «медленному» ОЗУ при записи и чтении операндов.

Таблица состоит из фиксированного числа строк. Каждая строка имеет следующие поля:

Адрес	Значение	Признак изменения	Код стирания

- адрес – физический адрес машинного слова в ОЗУ;
- значение – значение машинного слова, соответствующего адресу;
- признак изменения – код, характеризующий факт изменения поля значения (в соответствующей ячейке ОЗУ значение отличается от значения в таблице);
- код старения – код, характеризующий интенсивность обращений к данной строке. По значению поля определяются наиболее «популярные» строки. Конкретный алгоритм изменения данного поля зависит от ЭВМ.

Примерные алгоритмы использования буфера операндов

Алгоритм для чтения данных из ОЗУ

Пусть имеется команды чтения данных из машинного слова по физическому адресу $А_{исп}$.

1. Поиск по таблице строки, содержащей адрес, совпадающий с $А_{исп}$. Если такой строки нет, то на п. 3.
2. Происходит обновление кода старения. Результатом команды чтения является содержимое поля «Значение».
3. По значениям поля «Код старения» осуществляется поиск строки, используемой наименее интенсивно.
4. Анализируется код изменения. Если значение изменялось в таблице, то происходит запись значения по адресу в ОЗУ.
5. Считывается машинное слово из ОЗУ по адресу $А_{исп}$ и заполняется данная строка.
6. Считанное значение является результатом выполнения команды чтения.

Алгоритм для записи данных в ОЗУ

Пусть имеется команда записи значения в машинное слово по физическому адресу $А_{исп}$.

1. Поиск по таблице строки, содержащей адрес, совпадающий с $А_{исп}$. Если такой строки нет, то на п. 3.
2. Значение записывается в поле «Значение». Происходит обновление полей «Признак изменения», «Код старения». Выполнения команды записи завершено.
3. По значению поля «Код старения» осуществляется поиск строки, используемое наименее интенсивно.
4. Анализируется код изменения. Если значение изменялось в таблице, то происходит запись значения по адресу в ОЗУ.
5. Происходит обновление содержимого полей строки в соответствии с командой записи. Выполнение команды завершено.

3.3.2 Буферизация выборки команд

Буфер команд – минимизация обращений в ОЗУ за машинными командами.

Адрес	Значение	Код старения

Интерпретация одноименных полей аналогична буферу операндов.

Примерный алгоритм использования

Центральному процессору требуется для выполнения машинная команда, размещенная по физическому адресу ОЗУ $А_{исп}$.

1. Поиск по таблице строки, содержащей $А_{исп}$. Если такой не то на п. 3.
2. Обновление поля «Код старения», чтение поля «Значение» и передача его процессору для исполнения.
3. Поиск наименее интенсивно используемой строки. Чтение машинного слова из ОЗУ по адресу $А_{исп}$ и заполнение всех полей строки. Передача процессору значения для исполнения.

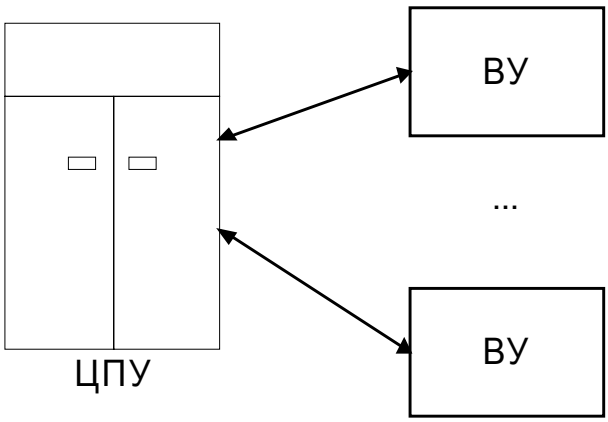
Конкретные реализация и алгоритмы зависят от архитектуры ЭВМ. Возможно, например, использование одного буфера.

Некоторые итоги решения проблемы оптимизации доступа к ОЗУ

3.4 Внешние устройства

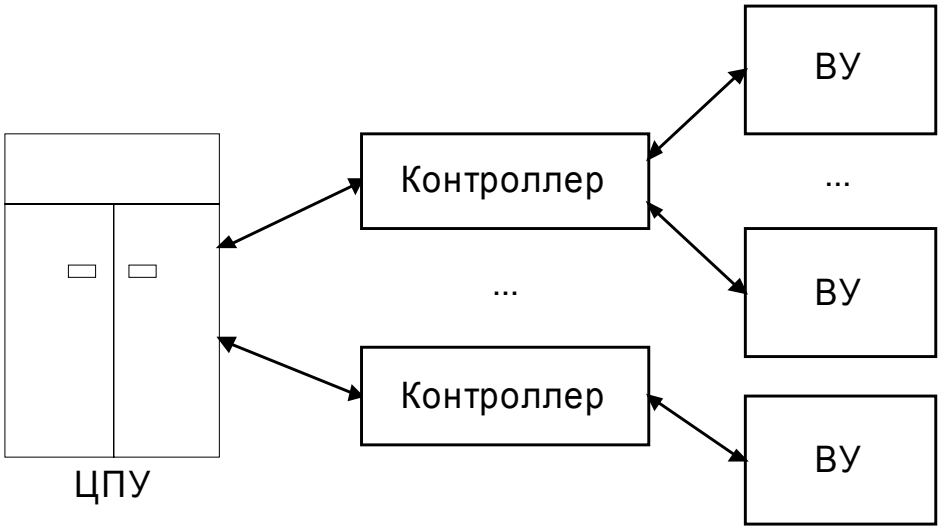
3.4.1 Организация управления внешними устройствами

Управление внешними устройствами с использованием специальных команд и/или специальных регистров внешних устройств (ВУ).
Возможно использование «специальных» областей ОЗУ для обмена информацией с внешними устройствами.
а) *Центральный процессор – ВУ*

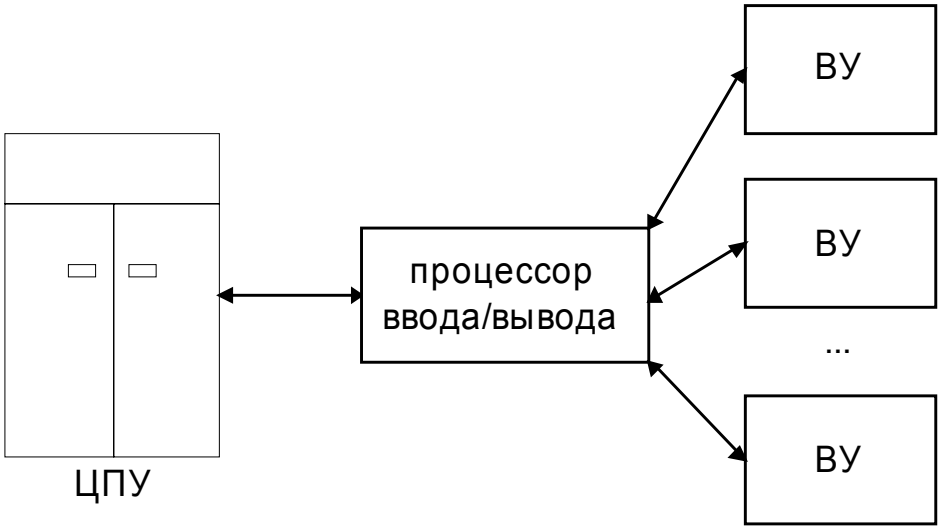


На ЦП ложатся все функции по управлению устройством. В этом случае осуществляется «микро» управление ВУ центральным процессором. Данная схема является простейшей, но объем затрат ЦП неоправданно велик (сложность соответствующих драйверов).

б) *ЦП – контроллер ВУ – ВУ*

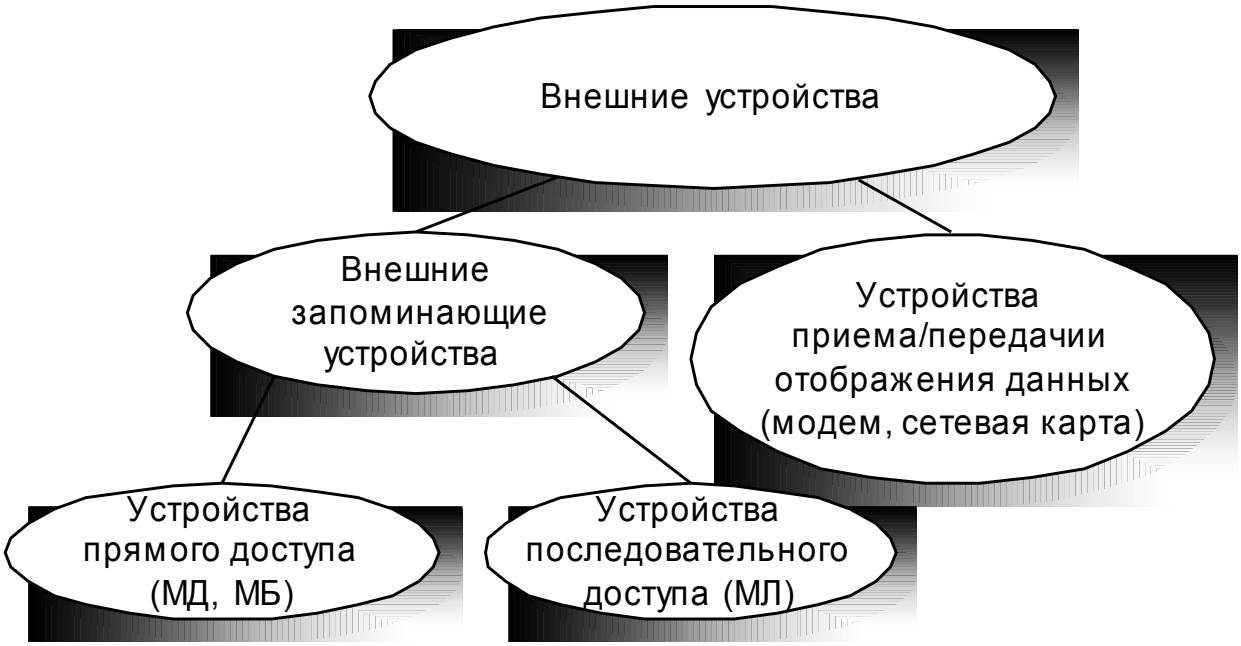


в) *ЦП – процессор ввода/вывода – ВУ*



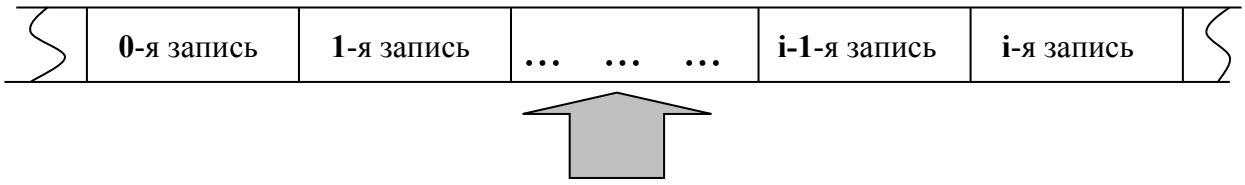
Для перечисленных вариантов ЦП использует запросы от уровня микрокоманд управления ВУ в случае *а)* до высокоуровневых макрозапросов в случае *б) и в)*.

3.4.2 Типы внешних устройств (ВУ)



Обмен с ВЗУ происходит некоторыми порциями данных – записями. Размер физической записи зависит от типа конкретного устройства. Все записи, размещенные на конкретном ВЗУ можно однозначно пронумеровать.

Устройства последовательного доступа.
ВЗУ является устройством последовательного доступа если для чтения i-й записи необходимо прочесть («просмотреть») предыдущие i-1 запись.
Примером устройства последовательного типа является магнитная лента (МЛ).



Обычно длина физической записи МЛ произвольная, она определяется специальными маркерами начала и конца записи.

Устройства последовательного доступа являются простейшими ВЗУ. Они обычно используются для архивирования данных. Скорость обработки запросов чтения/записи самая низкая (большой объем механических действий, таких как перемотка лент вперед-назад при выполнении обмена).

Устройства прямого доступа
Устройство прямого доступа характеризуется возможностью чтения любой записи без предварительного просмотра каких-либо других записей, размещенных на данном устройстве. ВЗУ прямого доступа классифицируются по производительности.

Магнитные диски (МД)
Наименее скоростные устройства прямого доступа. При выполнении обмена совершаются следующие действия:

- перемещение считывающей/головки на нужный цилиндр;
- ожидание выхода головки на начало нужного сектора диска (ожидание механического поворота диска на начало сектора);
- непосредственный обмен (в темпе движения диска);

Магнитный барабан

Высокоскоростное ВЗУ. При выполнении обмена совершаются следующие действия:

- электронное включение считывающей/записывающей головки, соответствующей нужному треку;
- ожидание размещения головки над началом нужного сектора (ожидание механического поворота барабана на начало сектора);
- непосредственный обмен в темпе движения барабана.
- магнитный барабан используется операционными системами высокопроизводительных ЭВМ для хранения оперативных данных (данных, время доступа к которым должно быть минимальным).

Матрично-электронные ВЗУ прямого доступа

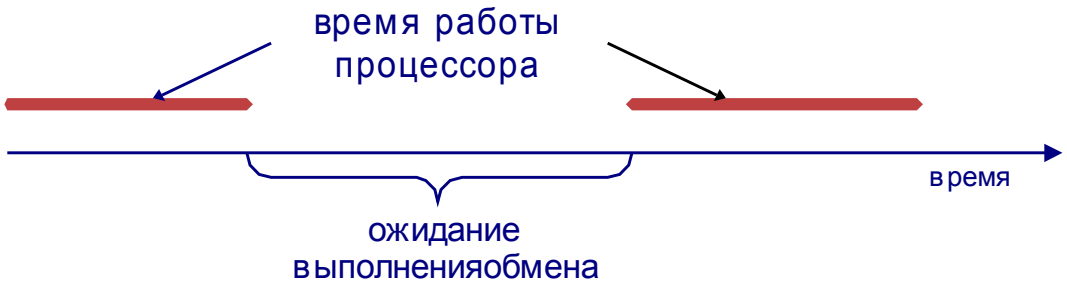
Память на магнитных доменах, ВЗУ, построенные на элементной базе ОЗУ и т.п.
В обмене отсутствует «механическая» составляющая, поэтому это наиболее быстродействующие ВЗУ.

3.4.3 Синхронная/асинхронная работа с ВУ

Существует две принципиально различные стратегии выполнения обмена с внешними устройствами: ***синхронная и асинхронная работа с ВУ.***

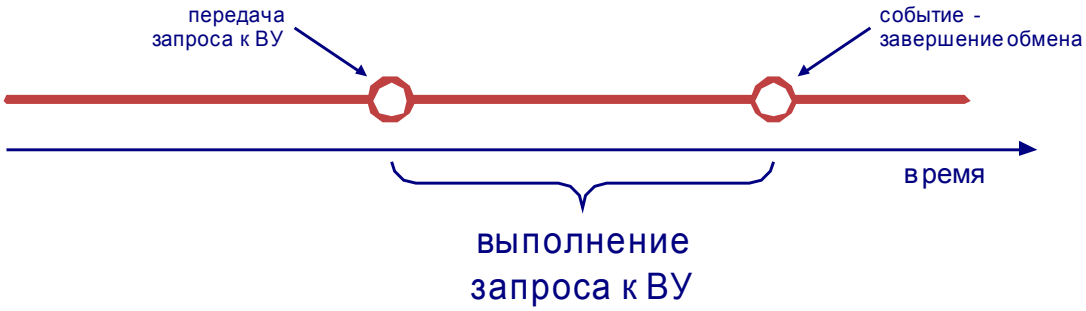
Синхронная работа с ВУ

Процессор подает запрос внешнему устройству и ожидает завершения выполнения запроса.



Системы с синхронной организацией работы ВУ неэффективны с точки зрения использования времени работы центрального процессора. Процессор часто «ожидает» выполнения запроса. Наиболее подходит для однопрограммных специализированных вычислительных систем.

Асинхронная работа с ВУ



При обработке запроса к ВУ происходит разделение выполнения на три части:
Первая – передача ЦП запроса на выполнение работ. После этого процессор может выполнять другие команды.
Вторая – параллельно работе ЦП происходит выполнение запроса к ВУ (т.е. в это время процессор может выполнять другие машинные команды).
Третья – выполнение работы ЦП прерывается и ему передается информация о завершении выполнения запроса. Следует отметить, что ЦП может также приостановить работу в случае обращения в область ОЗУ, находящуюся в обмене.

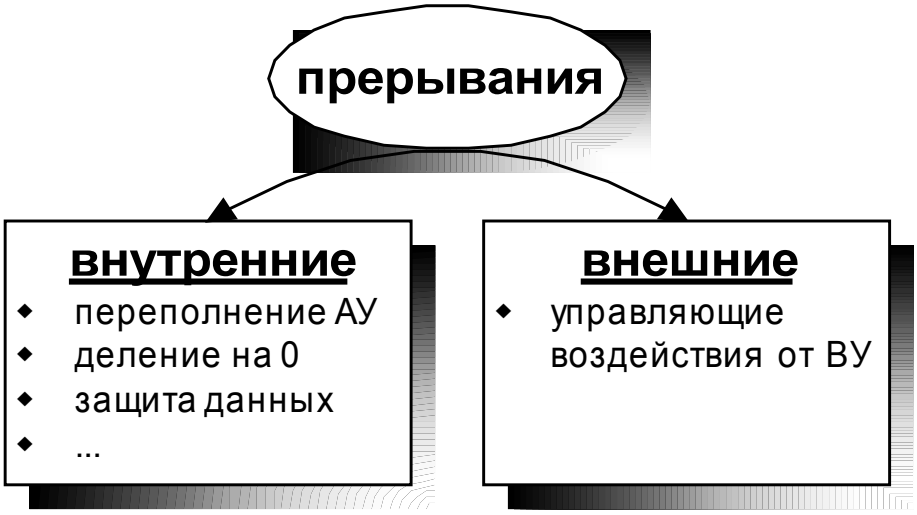
Асинхронная организация работы с ВУ более эффективна, но требует наличия развитого аппарата прерываний.

3.5 Аппарат прерываний

Аппарат прерываний ЭВМ - возможность аппаратуры ЭВМ стандартным образом обрабатывать возникающие в вычислительной системе события. Данные события будем называть прерываниями.

3.5.1 Определение. Последовательность действий при обработке

Итак, ***прерывание*** - одно из событий в вычислительной системе, на возникновение которого предусмотрена стандартная реакция аппаратуры ЭВМ. Количество различных типов прерываний ограничено и определяется при разработке аппаратуры ЭВМ.

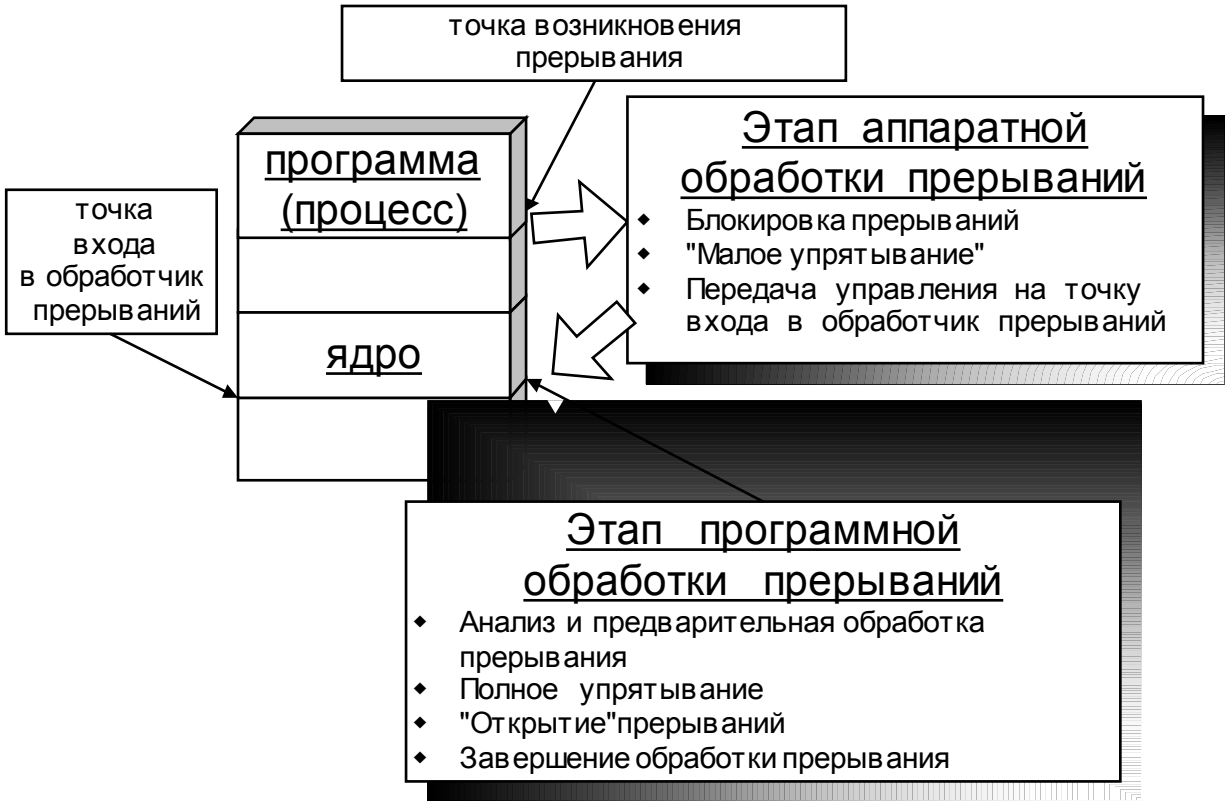


Прерывания можно разделить на две группы внутренние и внешние.
Внутренние прерывания инициируются схемами контроля работы процессора. Это может быть реакция ЦП на программную ошибку. Например, деление на ноль.

Внешние прерывания – это средство, позволяющее ЭВМ корректно взаимодействовать с внешними устройствами. Это может быть событие, связанное с поступлением новой информации от ВУ или возникновение ошибки во ВУ.

Аппарат прерываний ЦП обеспечивает стандартную реакцию аппаратуры при возникновении прерывания. Тем самым обеспечивается возможность корректной обработки прерываний в ВС.

Рассмотрим обобщенную (и упрощенную) модель последовательности действий, происходящих в ВС при возникновении прерывания.



При обработке события, связанного с возникновением прерывания на первом этапе работает аппаратура ВС. При этом аппаратно (без участия программы) выполняются следующие действия:

1. Включается режим блокировки прерываний. При этом режиме в системе запрещается инициализация новых прерываний. Возникающие в это время прерывания могут либо игнорироваться, либо откладываться (зависит от конкретной аппаратуры ЭВМ и типа прерывания).
2. Обработка прерывания предполагает сохранение возможности корректного продолжения прерванной программы (процесса) с точки прерывания. Поэтому следующий шаг аппаратной обработки – “малое упрятывание“ – копирование в специальную регистровую память ЦП (регистровый буфер, таблицу) минимального количества регистров и настроек ЦП, достаточных для запуска программы ОС, обрабатывающей прерывания. Это заведомо счетчик команд, регистр результата, некоторое количество регистров общего назначения. Следует отметить, что возможно организовать копирование (или упрятывание) всех регистров, используемых программой, но это, в общем случае, нецелесообразно, так как может потребовать значительных объемов регистровой памяти, а также потребует значительного времени работы в режиме блокировки прерываний.
3. Следующим шагом является переход на программный режим обработки прерываний. Для этих целей в аппаратуру ВМ обычно жестко “зашивается” адрес точки в ОЗУ, начиная с которой предполагается размещение части ОС, занимающейся программной обработкой прерываний – точка входа в обработчик прерываний. (Возможно определение не одной, а группы таких точек – по одной на тип или группу прерываний). Переход на программный этап обработки прерываний есть передача управления на точку входа в обработчик прерываний. Этот переход осуществляется не программно (за счет исполнения одной из команд передачи управления), а аппаратно (например, аппаратной записью в счетчик команд адреса точки входа).

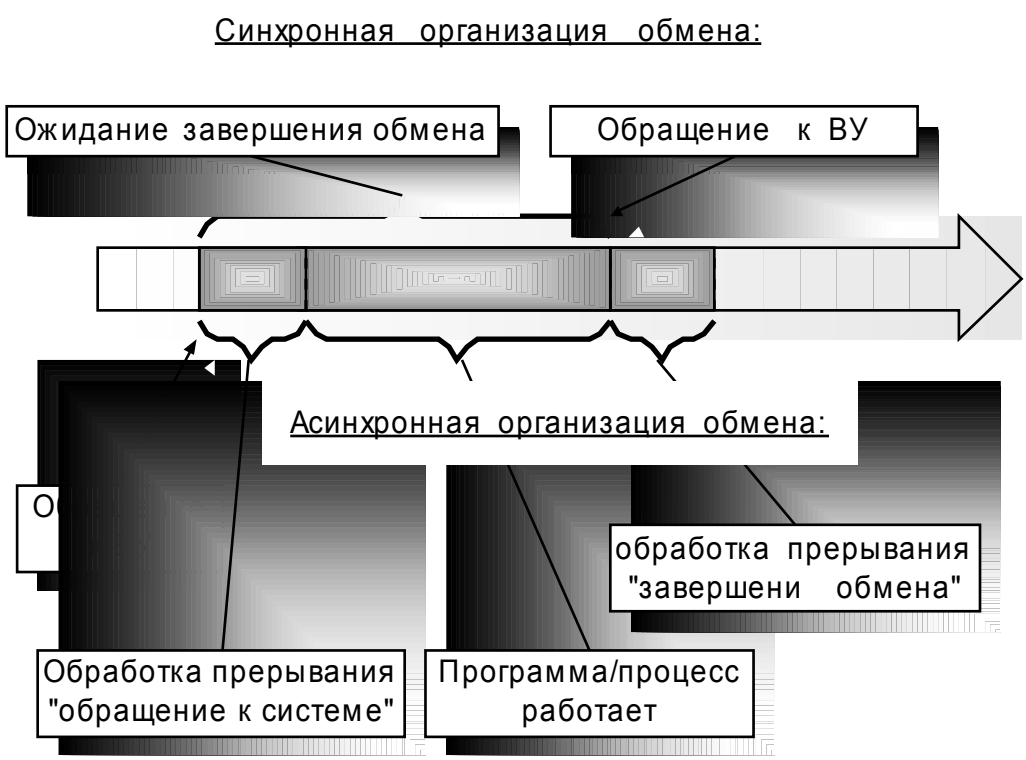
Второй этап. Программная обработка прерывания. Управление передано на точку ОС, занимающуюся обработкой прерывания. При входе в эту точку часть ресурсов ЦП, используемых программами освобождена (результат малого упрятывания). Поэтому будет запущена программа ОС, которая может использовать только освобожденные малым упрятыванием ресурсы ЦП (перечень доступных в этот момент ресурсов – характеристика аппаратуры). Выполняется следующая последовательность действий:

1. Анализ и предварительная обработка прерывания. Происходит идентификация типа прерывания, определяются причины. Если прерывание " короткое" обработка не требует дополнительных ресурсов ЦП и времени, то прерывание обрабатывается, происходит разблокировка прерываний и возврат в первоначальную программу (не вдаваясь в подробности, эта последовательность действий организована так, что гарантируется корректное восстановление всех регистров и настроек ЦП).
   Если прерывание требует использования всех ресурсов ЦП, то переходим к следующему шагу.
2. “Полное упрятывание” осуществляется полное упрятывание состояния всех ресурсов ЦП, использовавшихся прерванной программой (все регистры, настройки, режимы и т.д.) в специальную программную таблицу (в контекст процесса или программы – о нем позже). То есть в данную таблицу копируется содержимое аппаратной таблицы, содержащей сохраненные значения ресурсов ЦП после малого упрятывания, а также копируются все оставшиеся ресурсы ЦП используемые программно, но не сохраненные при малом упрятывании. После данного шага программе обработки прерываний становятся доступны все ресурсы ЦП, а прерванная программа получает статус ожидания завершения обработки прерывания. В общем случае программ или процессов, ожидающих завершения обработки прерывания может быть произвольное количество.
3. До данного момента времени все действия происходили в режиме блокировки прерываний. Почему? Потому что режим блокировки прерываний – единственная гарантия оттого, что не придет новое прерывание и при его обработке не потеряются данные, необходимые для продолжения прерванной программы (регистры, режимы, таблицы ЦП).
   После полного упрятывания разблокируются прерывания (то есть включается стандартный режим при котором возможно появление прерываний).
4. Заключительный этап – завершение обработки прерывания.

Вот упрощенная схема обработки прерывания, в реальных системах она может иметь отличия и быть сложнее. Но основные идеи обычно остаются неизменными. Аппарат прерываний позволяет системе фиксировать и корректно обрабатывать различные события, возникающие как внутри системы, так вне нее.

3.5.2 Прерывания: организация работы внешних устройств.

Одно из основных достижений прерываний – возможность организации асинхронной работы с внешними устройствами. Вернемся к ее рассмотрению. Пусть в системе имеется прерывание “обращение к системе”. Оно используется для организации доступа к функциям ОС. Рассмотрим рисунок:



Синхронная работа с ВУ

При синхронной организации обмена программа будет приостановлена с момента обращения к ВУ до момента завершения обмена. Дисбаланс между скоростью выполнения машинных команд и скоростью работы ВУ колоссальный. Поэтому задержки при синхронной работе крайне и крайне ощутимы.

Асинхронная работа с ВУ

Последовательность действий следующая

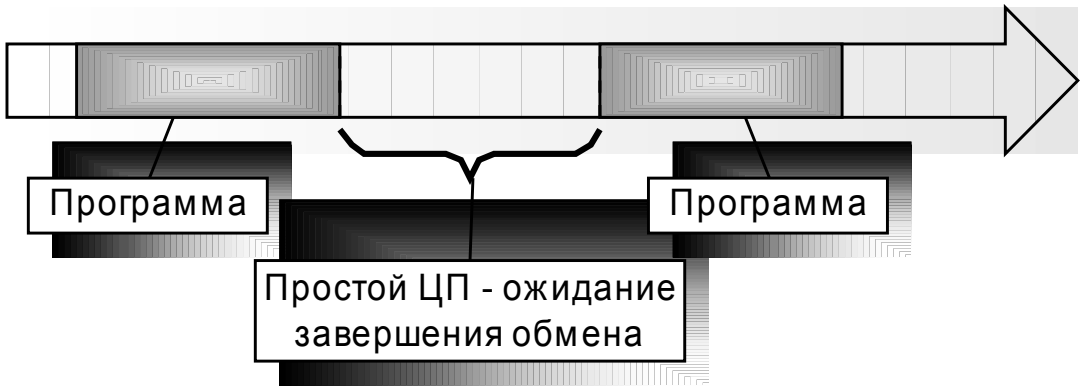
1. Программа инициирует прерывание “обращение к системе”, тем самым передается заказ на выполнение обмена, (параметры заказа могут быть переданы через специальные регистры, стек и т.п.) Происходит обработка прерывания (при этом программа (процесс) находится в ожидании). При обработке прерывания конкретному драйверу устройства передается заказ на выполнение обмена (который поступает в очередь).
2. После завершения обработки прерывания “обращение к системе” программа продолжает свое выполнение до завершения обмена (на самом деле это не всегда так, почему – ответ позднее).
3. Выполнение программы приостанавливается по причине возникновебния прерывания – завершение обмена с конкретным устройством. После обработки прерывания выполнение будет продолжено.

Очевидно, что асинхронная схема обработки обращений к ВУ позволяет сглаживать системный дисбаланс в скорости выполнения машинных команд и скоростью доступа к ВУ. Это еще одно из решений объявленной в начале курса проблемы.

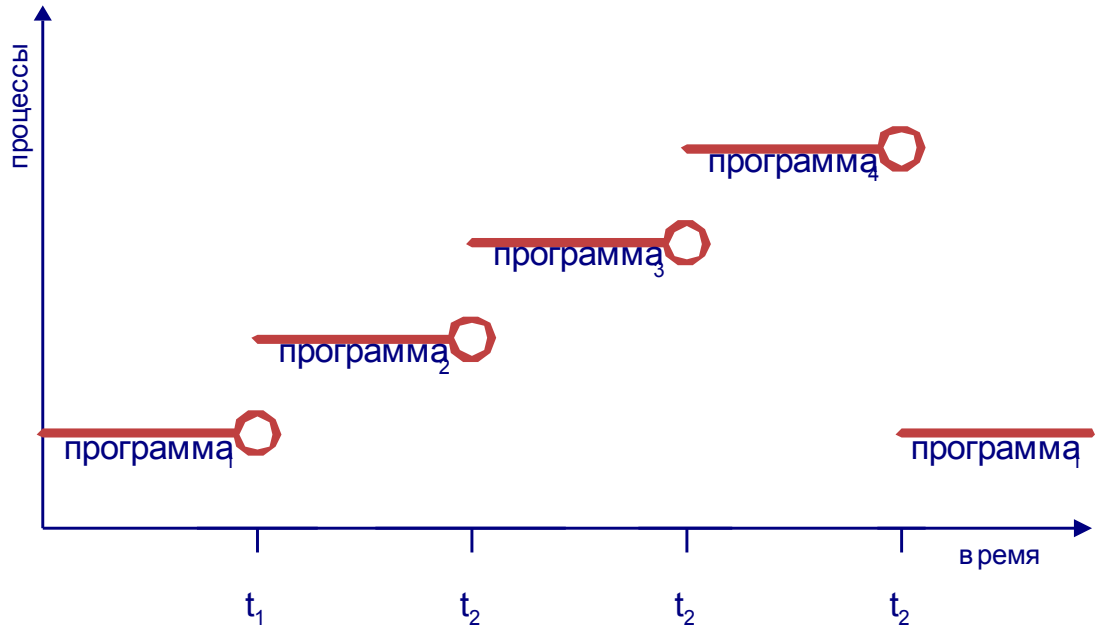
Представленная выше схема организации обмена является достаточно упрощенной. Она не затрагивает случаев синхронизации доступа к областям памяти, участвующим в обмене. Проблема состоит в том, что, например, записывая некую область данных на ВЗУ, после обработки заказа на обмен, но до завершения обмена, программа может попытаться обновить содержимое области, что является некорректным. Поэтому в реальных системах для синхронизации работы с областями памяти, находящимися в обмене, используется возможность ее аппаратного закрытия на чтение и/или запись. То есть при попытке обмена с закрытой областью памяти произойдет прерывание. Это позволяет остановить выполнение программы до завершения обмена, если программа попытается выполнить некорректные операции с областью памяти, находящейся в обмене (попытка чтения при незавершенной операции чтения с ВУ или записи при незавершенной операции записи данной области на ВУ).

3.6 Аппаратная поддержка мультипрограммирования

Итак, выше мы выяснили, что, несмотря на возможность асинхронной работы с ВУ, имеют место периоды ожидания программой завершения обмена. Если система обрабатывает единственную программу, то в это время ЦП не производит никакой полезной работы, то есть простаивает (на самом деле термин простой достаточно условный, так как при этом работает операционная система).



Решением проблемы простоя ЦП в этом случае является использование ВС в ***мультипрограммном режиме***, в режиме при котором возможна организация переключения выполнения с одной программы на другую



На рисунке изображена подобная мультипрограммная система, обрабатывающая одновременно 4 программы (процесса). t_1 – момент времени в который программа$_1$ будет остановлена для ожидания завершения обмена (до момента времени t_4). В момент времени t_1 система запускает выполнение программы$_2$, которая выполняется до момента времени t_2. С t_2 программа$_2$ также начинает ждать завершения своего обмена и т.д.

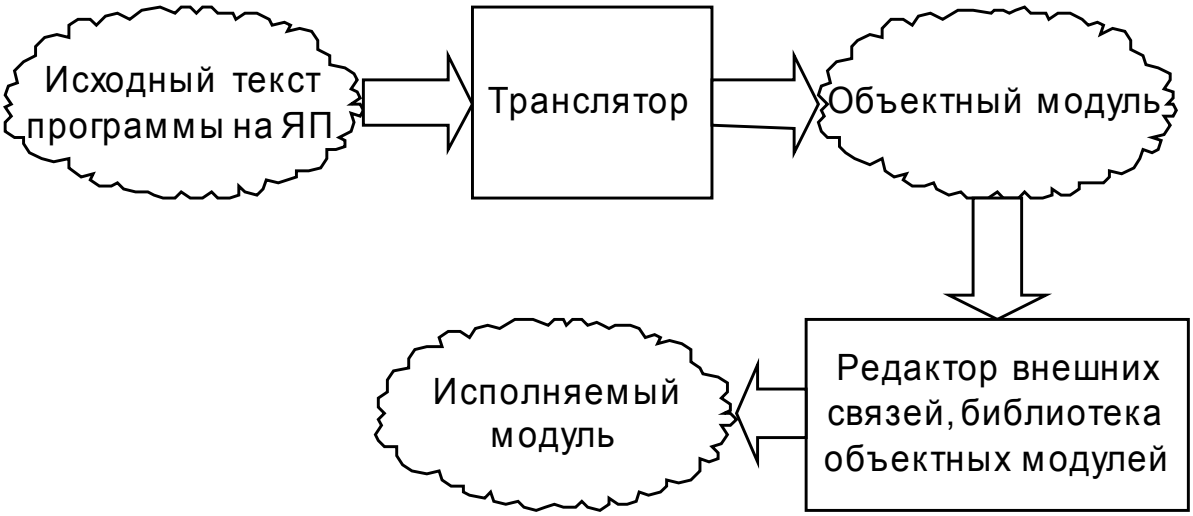
Для корректной организации мультипрограммной обработки необходима аппаратная поддержка ЭВМ. Как минимум аппаратура ЭВМ должна поддерживать следующие функции.

1. ***Аппарат защиты памяти***. Аппаратная возможность ассоциирования некоторых областей ОЗУ с одним из выполняющихся процессов/программ. Настройка аппарата защиты памяти происходит аппаратно, то есть назначение программе/процессу области памяти происходит программно (т.е., в общем случае операционная система устанавливает соответствующую информацию в специальных регистрах), а контроль за доступом – автоматически. При этом при попытке другим процессом/программой обратиться к этим областям ОЗУ происходит прерывание “Защита памяти”
2. Наличие специального режима ***операционной системы (привилегированный режимом или режим супервизора)*** ЦП. Суть заключается в следующем: все множество машинных команд разбивается на 2 группы. Первая группа – команды, которые могут исполняться всегда (пользовательские команды). Вторая группа – команды, которые могут исполняться только в том случае, если ЦП работает в режиме ОС. Если ЦП работает в режиме пользователя, то попытка выполнения специализированной команды вызовет прерывание – "Запрещенная команда”. Какова необходимость наличия такого режима выполнения команд? Простой пример – управление аппаратом защиты памяти. Для корректного функционирования этого аппарата необходимо обеспечить централизованный доступ к командам настройки аппарата защиты памяти. То есть эта возможность должна быть доступна не всем программам.
3. Необходимо наличие аппарата прерываний. Как минимум в машине должно быть прерывание по таймеру, что позволит избежать “зависания“ всей системы при зацикливании одной из программ.

3.7 Аппарат виртуальной памяти

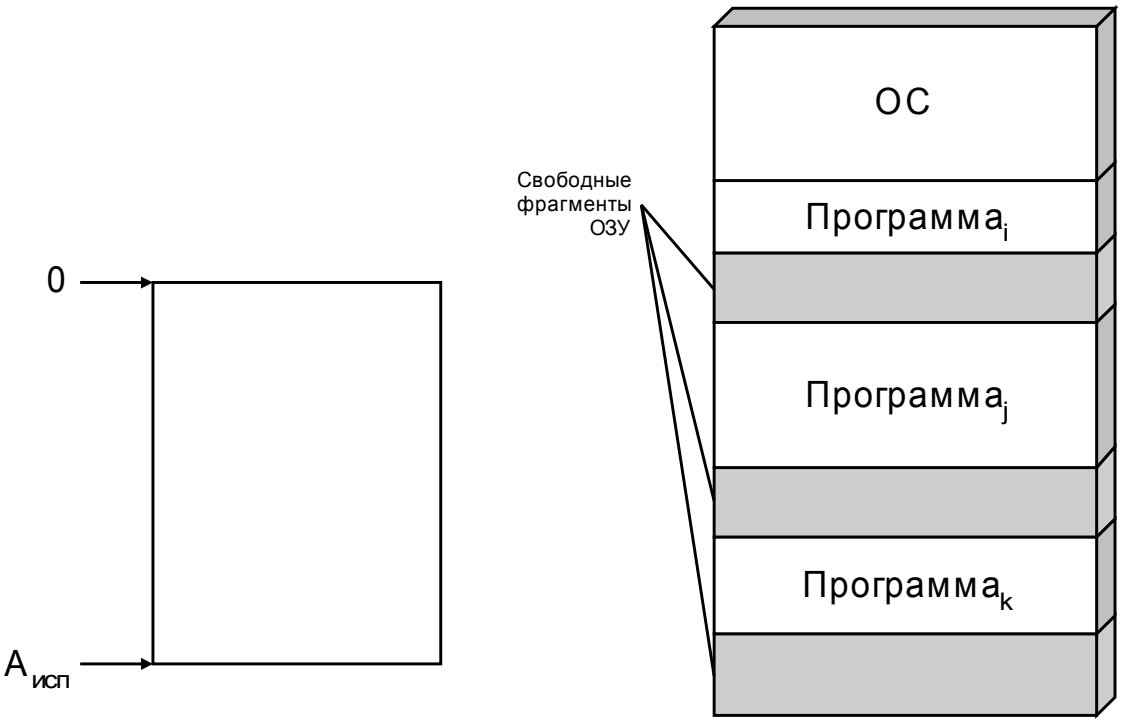
Рассмотрим некоторые проблемы организации адресации в программах/процессах и связанные с ними проблемы использования ОЗУ в целом.

В общем случае схема получения исполняемого кода программы следующая:



Данная схема достаточно очевидна, так как она связана с привычным для нас процессами трансляции. Остановимся подробней на исполняемом модуле. Данный модуль представляет собой готовую к выполнению программу в машинных кодах. При этом внутри программы к моменту образования исполняемого модуля используется модель организации адресного пространства программы (эта модель, в общем случае не связана с теми ресурсами ОЗУ, которые предполагается использовать позднее). Для простоты будем считать, что данная модель представляет собой непрерывный фрагмент адресного пространства в пределах которого размещены данные и команды программы. Будем называть подобную организацию адресации в ***программе программной адресацией или логической/виртуальной адресацией***. Итак, повторяем, на уровне исполняемого кода имеется программа в машинных кодах, использующая адреса данных и команд. Эти адреса в общем случае не являются адресами конкретных физических ячеек памяти, в которых размещены эти данные, более того, в последствии мы увидим, что виртуальным (или программным) адресам могут ставиться в соответствие произвольные физические адреса памяти. То есть при реальном исполнении программы далеко не всегда виртуальная адресация, используемая в программе совпадает с физической адресацией, используемой ЦП при выполнении данной программы.

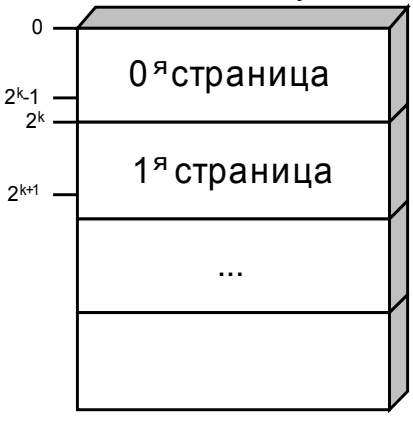
Элементарное программно-аппаратное решение – использование возможности ***базирования адресов***. Суть его состоит в следующем: пусть имеется исполняемый программный модуль. Виртуальное адресное пространство этого модуля лежит в диапазоне [0, $A_{кон}$]. В ЭВМ выделяется специальный регистр базирования $R_{баз.}$, который содержит физический адрес начала области памяти, в которой будет размещен код данного исполняемого модуля. При этом исполняемые адреса, используемые в модуле будут автоматически преобразовываться в адреса физического размещения данных путем их сложения с регистром $R_{баз.}$. Таким образом код используемого модуля может перемещаться по пространству физического ОЗУ. Эта схема является элементарным решением организации простейшего ***аппарата виртуальной памяти***. То есть аппарата, позволяющего автоматически преобразовывать ***виртуальные адреса программы*** в адреса физической памяти.
Рассмотрим более сложные механизмы организации виртуальной памяти.

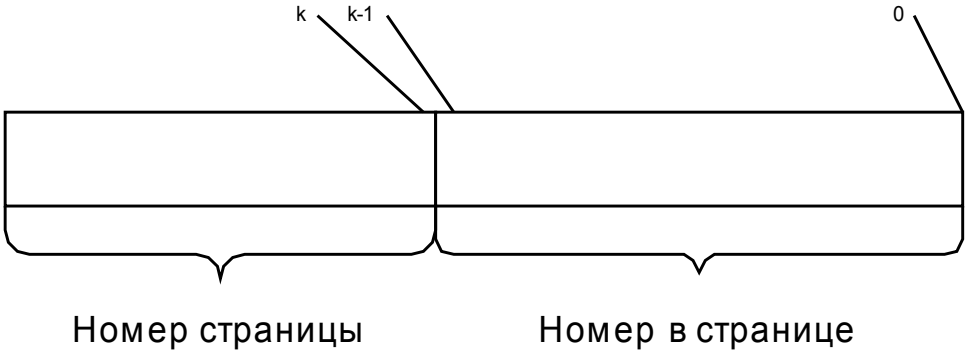


Пусть имеется вычислительная система, функционирующая в мультипрограммном режиме. То есть одновременно в системе обрабатываются несколько программ/процессов. Один из них занимает ресурсы ЦП. Другие ждут завершения операций обмена, третьи – готовы к исполнению и ожидают предоставления ресурсов ЦП. При этом происходит завершение выполнявшихся процессов и ввод новых, это приводит к возникновению проблемы фрагментации ОЗУ. Суть ее следующая. При размещении новых программ/процессов в ОЗУ ЭВМ (для их мультипрограммной обработки) образуются свободные фрагменты ОЗУ между программами/процессами. Суммарный объем свободных фрагментов может быть достаточно большим, но, в то же время, размер самого большого свободного фрагмента недостаточно для размещения в нем новой программы/процесса. В этой ситуации возможна деградация системы – в системе имеются незанятые ресурсы ОЗУ, но они не могут быть использованы. Путь решения этой проблемы – использование более развитых механизмов организации ОЗУ и виртуальной памяти, позволяющие отображать виртуальное адресное пространство программы/процесса не в одну непрерывную область физической памяти, а в некоторую совокупность областей.

Организация страничной памяти

Страничная организация памяти предполагает разделение всего пространства ОЗУ на блоки одинакового размера – страницы. Обычно размер страницы равен 2^k. В этом случае адрес, используемый в данной ЭВМ, будет иметь следующую структуру:





Модельная (упрощенная) схема организации функционирования страничной памяти ЭВМ следующая: Пусть одна система команд ЭВМ позволяет адресовать и использовать m страниц размером 2^k каждая. То есть виртуальное адресное пространство программы/процесса может использовать для адресации команд и данных до m страниц.

Физическое адресное пространство, в общем случае может иметь произвольное число физических страниц (их может быть больше m, а может быть и меньше). Соответственно структура исполнительного физического адреса будет отличаться от структуры исполнительного виртуального адреса за счет размера поля ”номер страницы”.

В виртуальном адресе размер поля определяется максимальным числом виртуальных страниц – m.

В физическом адресе – максимально возможным количеством физических страниц, которые могут быть подключены к данной ЭВМ (это также фиксированная аппаратная характеристика ЭВМ).

В ЦП ЭВМ имеется аппаратная таблица страниц (иногда таблица приписки) следующей структуры:

0	α_0
1	α_1
2	α_2
3	α_3
...	
i	α_i
...	
m-1	α_{m-1}

Таблица содержит m строк. Содержимое таблицы определяет соответствие виртуальной памяти физической для выполняющейся в данный момент программы/процесса. Соответствие определяется следующим образом: i-я строка таблицы соответствует i-й виртуальной странице.

Содержимое строки α_i определяет, чему соответствует i-я виртуальная страница программы/процесса. Если $\alpha_i \geq 0$, то это означает, что α_i есть номер физической страницы, которая соответствует виртуальной странице программы/процесса. Если $\alpha_i = -1$, то это означает, что для i-й виртуальной страницы нет соответствия физической странице ОЗУ (обработка этой ситуации ниже).

Итак, рассмотрим последовательность действий при использовании аппарата виртуальной страничной памяти.

1. При выполнении очередной команды схемы управления ЦП вычисляют некоторый адрес операнда (операндов) $А_{исп}$. Это виртуальный исполнительный адрес.
2. Из $А_{исп.}$ Выделяются значимые поля номер страницы (номер виртуальной страницы). По этому значению происходит индексация и доступ к соответствующей строке таблицы страниц.
3. Если значение строки ≥ 0, то происходит замена содержимого поля номер страницы на соответствующее значение строки таблицы, таким образом, получается физический адрес. И далее ЦП осуществляет работу с физическим адресом.
4. Если значение строки таблицы равно –1 это означает, что полученный виртуальный адрес не размещен в ОЗУ. Причины такой ситуации? Их две. Первая – данная виртуальная страница отсутствует в перечне станиц, доступных для программы/процесса, то есть имеет место попытка обращения в “ чужую”, не легитимную память. Вторая ситуация, когда операционная система в целях оптимизации использования ОЗУ, откачала некоторые страницы программы/процесса в ВЗУ(свопинг, при действиях ОС при свопинге позднее). Что происходит в системе, если значение строки таблицы страниц –1, и мы обратились к этой строке? Происходит прерывание “ защита памяти”, управление передается операционной системе (по стандартной схеме обработки прерывания и далее происходит программная обработка ситуации (обращаем внимание, что все, что выполнялось до сих пор – пункт 1, 2, 3 и 4 – это действия аппаратуры, без какого-либо участия программного обеспечения).
   ОС по содержимому внутренних данных определяет конечную причину данного прерывания: или это действительно защита памяти, или мы пытались обратиться к странице ОЗУ, которая временно размещена во внешней памяти.

Таким образом, предложенная модель организации виртуальной памяти позволяет решить проблему фрагментации ОЗУ. На самом деле, некоторая фрагментация остается (если в странице занят хотя бы 1 байт, то занята вся страница), но она является контролируемой и не оказывает значительного влияния на производительность системы.

Далее, данная схема позволяет простыми средствами организовать защиту памяти, а также своппирование страниц.

Предложенная модель организации виртуальной памяти позволяет иметь отображение виртуального адресного пространства программы/процесса в произвольные физические адреса, также позволяет выполнять в системе программы/процессы, размещенные в ОЗУ частично (оставшаяся часть может быть размещена во внешней памяти).

Недостаток – необходимость наличия в ЦП аппаратной таблицы значительных размеров.

Итак мы рассмотрели модельный, упрощенный вариант организации виртуальной памяти. Реальные решения используемые в различных архитектурах ЭВМ могут быть гораздо сложнее, но основные идеи остаются неизменными.

4 Операционная система. Базовые понятия, определения.

4.1 Базовые понятия, определения, структура

Операционная система – это комплекс программ, обеспечивающий контроль за существованием (некоторые из ресурсов ВС, как мы знаем, являются программными или логическими/виртуальными и создаются под контролем операционной системой), распределением и использованием ресурсов ВС.

Любая ОС оперирует некоторым набором базовых сущностей (понятий) на основе которых строится логика функционирования системы. Например, подобными базовыми понятиями могут быть задача, задание, процесс, набор данных, файл, объект.

Одним из наиболее распространенных базовых понятий ОС является ***процесс***.

Интуитивно определение процесса достаточно просто, но определить процесс строго, формально, достаточно сложно. Поэтому существует целый ряд определений процесса, многие из которых системно-ориентированы.

Процесс – это совокупность машинных команд и данных, исполняющаяся в рамках ВС и обладающая правами на владение некоторым набором ресурсов. Эти права могут быть эксклюзивными, когда ресурс принадлежит только этому процессу. Некоторые из ресурсов могут разделяться, т. е. одновременно принадлежать двум и более процессам, в этом случае мы говорим о ***разделяемых ресурсах***.

Возможно два варианта выделения ресурсов процессу:

- предварительная декларация использования тех или иных ресурсов (до начала выполнения процесса в систему передается перечень ресурсов, которые будут использованы процессом);
- Динамическое пополнение списка принадлежащих процессу ресурсов по ходу выполнения процесса при непосредственном обращении к ресурсу.

Реальная схема зависит от конкретной ОС. На практике возможно использование комбинации этих вариантов. Для простоты изложения будем считать, что модельная ОС имеет возможность предварительной декларации ресурсов, которые будут использованы процессом.

Любая ОС должна удовлетворять следующим свойствам:

- надежность
- защита
- эффективность
- предсказуемость

Типовая структура ОС.

Ядро – резидентная часть ОС, работающая в режиме супервизора. В ядре размещаются программы обработки прерываний и драйверы наиболее «ответственных» устройств. Это могут быть и физические, и виртуальные устройства. Например, в ядре могут располагаться драйверы файловой системы, ОЗУ. Обычно ядро работает в режиме физической адресации.

Следующие уровни структуры – динамически подгружаемые драйверы физических и виртуальных устройств. Это драйверы, добавление которых в систему возможно «на ходу» без перекомпоновки программ ОС. Они могут являться резидентными и нерезидентными, а также могут работать как в режиме супервизора, так и в пользовательском режиме.

Можно выделить следующие основные логические функции ОС:

- управление процессами;
- управление ОП;
- планирование;
- управление устройствами и ФС.

4.2 Управление процессами

4.2.1 Жизненный цикл процесса

Рассмотрим типовые этапы обработки процесса в системе, совокупность этих этапов будем назвать ***жизненным циклом процесса*** в системе. Традиционно, жизненный цикл процесса содержит этапы:

- образование (порождение) процесса;
- обработка (выполнение) процесса;
- ожидание (по тем или иным причинам) постановки на выполнение;
- завершение процесса.

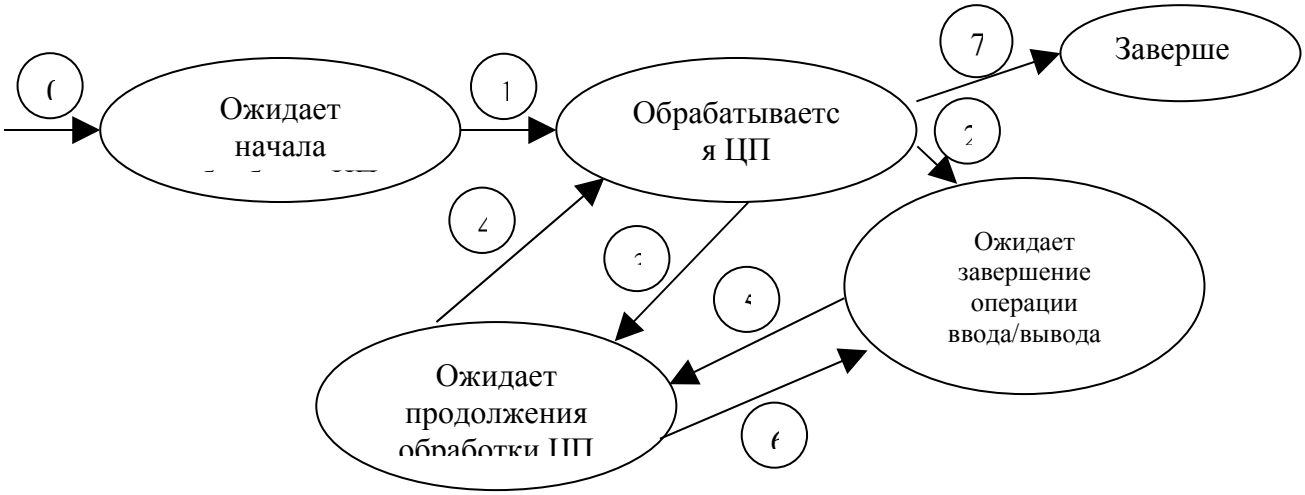
Однако жизненные циклы процессов в реальных системах могут иметь свою, системно-ориентированную совокупность этапов.

Рассмотрим модельную ОС.

Пусть имеется специальный ***буфер ввода процессов*** (БВП) – пространство, в котором размещаются и хранятся сформированные процессы от момента их образования, до момента начала выполнения. На данном этапе происходит формирование всех необходимых структур данных, соответствующих процессу. В частности, на данном этапе ОС формирует информацию о предварительно заказанных данным процессом ресурсах. Основная задача БВП – «подпитка» системы новыми процессами, готовыми к исполнению.

После начала выполнения процесса он попадает в ***буфер обрабатываемых процессов*** (БОП). В данном буфере размещаются все процессы, находящиеся в системе в мультипрограммной обработке.

Обобщенный жизненный цикл процесса можно представить в этом случае ***графом состояний***. Рассмотрим, кратко, переходы процесса из состояния в состояние.



0. После формирования процесс поступает в очередь на начало обработки ЦП (попадает в БВП).
1. В БВП выбирается наиболее приоритетный процесс для начала обработки ЦП (попадает в БОП).
2. Процесс прекращает обработку ЦП по причине ожидания операции в/в, поступает в очередь завершения операции обмена (БОП).
3. Процесс прекращает обработку ЦП, но в любой момент может быть продолжен (например, истек квант времени ЦП, выделенный процессу). Поступает в очередь процессов, ожидающих продолжения выполнения центральным процессором (БОП).
4. Наиболее приоритетный процесс продолжает выполнение ЦП (БОП).
5. Операция обмена завершена и процесс поступает в очередь ожидания продолжения выполнения ЦП (БОП).
6. Переход из очереди готовых к продолжению процессов в очередь процессов, ожидающих завершения обмена (например, ОС откачала содержимое адресного пространства процесса из ОЗУ во внешнюю память) (БОП).
7. Завершение процесса, освобождение системных ресурсов. Корректное завершение работы процесса, разгрузка информационных буферов, освобождение ресурсов (например, реальный вывод информации на устройство печати).

Текущее состояние любого процесса из БОП изменяется во времени в зависимости от самого процесса и состояния ОС. С каждым из процессов из БОП система ассоциирует совокупность данных, характеризующих актуальное состояние процесса – ***контекст процесса***. (в общем случае контекст процесса содержит информацию о текущем состоянии процесса, включая информацию о режимах работы процессора, содержимом регистровой памяти, используемой процессом, системной информации ОС, ассоциированной с данным процессом).

Процессы, находящиеся в одном из состояний ожидания в своих контекстах содержат всю информацию, необходимую для продолжения выполнения - состояние процесса в момент прерывания (копии регистров, режимы ОП, настройки аппарата виртуальной памяти и т. д.). Соответственно при смене выполняемого процесса ОС осуществляет «перенастройку» внутренних ресурсов ЦП, происходит смена контекстов выполняемых процессов.
На этапе выполнения процесса ОС обеспечивает возможность корректного взаимодействия процессов от передачи сигнальных воздействий от процесса к процессу до организации корректной работы с разделяемыми ресурсами.

Итак, контекст процесса может состоять из:

- пользовательской составляющей – состояние программы, как совокупности машинных команд и данных, размещенных в ОЗУ;
- системной составляющей – содержимое регистров и режимов работы процессора, настройки аппарата защиты памяти, виртуальной памяти, принадлежащие процессу ресурсы (как физические, так и виртуальные).

4.2.2 Типы процессов

В различных системах используются различные трактовки определения термина ***процесс***. Рассмотрим уточнение понятия процесса.

Полновесные процессы - это процессы, выполняющиеся внутри защищенных участков памяти операционной системы, то есть имеющие собственные виртуальные адресные пространства для статических и динамических данных. В мультипрограммной среде управление такими процессами тесно связано с управлением и защитой памяти, поэтому переключение процессора с выполнения одного процесса на выполнение другого является достаточно дорогой операцией. В дальнейшем, используя термин ***процесс*** будем подразумевать ***полновесный процесс***.

Легковесные процессы, называемые еще как ***нити*** или ***сопрограммы***, не имеют собственных защищенных областей памяти. Они работают в мультипрограммном режиме одновременно с активировавшей их задачей и используют ее виртуальное адресное пространство, в котором им при создании выделяется участок памяти под динамические данные (стек), то есть они могут обладать собственными локальными данными. Нить описывается как обычная функция, которая может использовать статические данные программы. Для одних операционных систем можно сказать, что нити являются некоторым аналогом процесса, а в других нити представляют собой части процессов. Таким образом, обобщая можно сказать – в любой операционной системе понятие «процесс» включает в себя следующее:

- исполняемый код;
- собственное адресное пространство, которое представляет собой совокупность виртуальных адресов, которые может использовать процесс;
- ресурсы системы, которые назначены процессу ОС;
- хотя бы одну выполняемую нить.

При этом подчеркнем – понятие процесса может включать в себя понятие исполняемой нити, т. е. однонитевую организацию – «один процесс – одна нить». В данном случае понятие процесса жестко связано с понятием отдельной и недоступной для других процессов виртуальной памяти. С другой стороны, в процессе может несколько нитей, т. е. процесс может представлять собой многонитевую организацию.

Нить также имеет понятие контекста – это информация, которая необходима ОС для того, чтобы продолжить выполнение прерванной нити. Контекст нити содержит текущее состояние регистров, стеков и индивидуальной области памяти, которая используется подсистемами и библиотеками. Как видно, в данном случае характеристики нити во многом аналогичны характеристикам процесса. С точки зрения процесса, нить можно определить как независимый поток управления, выполняемый в контексте процесса. При этом каждая нить, в свою очередь, имеет свой собственный контекст.

4.2.3 Взаимодействие процессов. Методы синхронизации

Процессы, выполнение которых хотя бы частично перекрывается по времени, называются параллельными. Они могут быть независимыми и взаимодействующими. ***Независимые процессы*** – процессы, использующие независимое множество ресурсов и на результат работы такого процесса не влияет работа независимого от него процесса. Наоборот – ***взаимодействующие процессы*** совместно используют ресурсы и выполнение одного может оказывать влияние на результат другого.

Совместное использование несколькими процессами ресурса ВС, когда каждый из процессов одновременно владеет ресурсом называют **разделением ресурса**. Разделению подлежат как аппаратные, так программные ресурсы. Разделяемые ресурсы, которые должны быть доступны в текущий момент времени только одному процессу – это так называемые **критические ресурсы**. Таковыми ресурсами могут быть, как внешнее устройство, так и некая переменная, значение которой может изменяться разными процессами.

Необходимо уметь решать две важнейшие задачи:

1. Распределение ресурсов между процессами.
2. Организация защиты адресного пространства и других ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов.

Важнейшим требованием мультипрограммирования с точки зрения распределения ресурсов является следующее: результат выполнения процесса не должен зависеть от порядка переключения выполнения между процессами, т.е. от соотношения скорости выполнения процесса со скоростями выполнения других процессов.

Рассмотрим ситуацию, изображенную на Рис. 1:

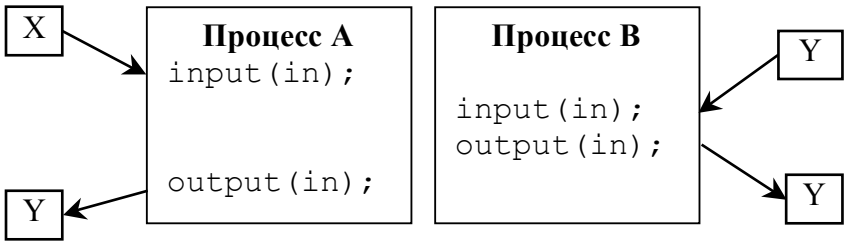


Рис. 1 Конкуренция процессов за ресурс.

В этом случае символ, считанный процессом **А**, был потерян, а символ, считанный процессом В, был выведен дважды. Результат выполнения процессов здесь зависит от того, в какой момент осуществляется переключение процессов, и от того, какой конкретно процесс будет выбран для выполнения следующим. Такие ситуации называются **гонками** (race conditions) между процессами, а процессы – конкурирующими. Единственный способ избежать гонок при использовании разделяемых ресурсов – контролировать доступ к любым разделяемым ресурсам в системе. При этом необходимо организовать **взаимное исключение** – т.е. такой способ работы с разделяемым ресурсом, при котором постулируется, что в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

Проблему организации взаимного исключения можно сформулировать в более общем виде. Часть программы (фактически набор операций), в которой осуществляется работа с критическим ресурсом, называется **критической секцией**, или **критическим интервалом**. Задача взаимного исключения в этом случае сводится к тому, чтобы не допускать ситуации, когда два процесса одновременно находятся в критических секциях, связанных с одним и тем же ресурсом.

Заметим, что вопрос организации взаимного исключения актуален не только для взаимосвязанных процессов, совместно использующих определенные ресурсы для обмена информацией. Выше отмечалось, что возможна ситуация, когда процессы, не подозревающие о существовании друг друга, используют глобальные ресурсы системы, такие как устройства ввода/вывода, принтеры и т.п. В этом случае имеет место конкуренция за ресурсы, доступ к которым также должен быть организован по принципу взаимного исключения.

При организации взаимного исключения могут возникнуть **тупики (deadlocks**), ситуации в которой конкурирующие за критический ресурс процессы вступают в клинч – безвозвратно блокируются.

Есть два процесса **A** и **B**, каждому из которых в некоторый момент требуется иметь доступ к двум ресурсам $\mathbf{R_1}$ и $\mathbf{R_2}$**.** Процесс **A** получил доступ к ресурсу $\mathbf{R_1}$, и следовательно, никакой другой процесс не может иметь к нему доступ, пока процесс **A** не закончит с ним работать. Одновременно процесс **B** завладел ресурсом $\mathbf{R_2}$**.** В этой ситуации каждый из процессов ожидает освобождения недостающего ресурса, но оба ресурса никогда не будут освобождены, и процессы никогда не смогут выполнить необходимые действия.

Далее мы рассмотрим различные механизмы организации взаимного исключения для синхронизации доступа к разделяемым ресурсам и обсудим достоинства, недостатки и области применения этих подходов.

Рассмотри классические методы (средства) синхронизации.

4.2.3.1 Семафоры Дейкстры

Тип данных, именуемый семафором. Семафор представляет собой переменную целого типа **S**, над которой определены две операции: **down(s)** (или **P(S)**) и **up(S)** (или **V(S)**). Оригинальные обозначения **P** и **V**, данные Дейкстрой и получившие широкое распространение в литературе, являются сокращениями голландских слов *proberen* – проверить и *verhogen* – увеличить.

Операция **down(S)** проверяет значение семафора, и если оно больше нуля, то уменьшает его на **1**. Если же это не так, процесс блокируется, причем операция **down** считается незавершенной. Важно отметить, что вся операция является неделимой, т. е. Проверка значения, его уменьшение и, возможно, блокирование процесса производится как одно атомарное действие, которое не может быть прервано. Операция **up(S)** увеличивает значение семафора на **1**. При этом, если в системе присутствуют процессы, блокированные ранее при выполнении down на этом семафоре, ОС разблокирует один из них с тем, чтобы он завершил выполнение операции **down**, т. е. Вновь уменьшил значение семафора. При этом также постулируется, что увеличение значения семафора и, возможно, разблокирование одного из процессов и уменьшение значения являются атомарной неделимой операцией.

Чтобы прокомментировать работу семафора рассмотрим пример. Представим себе супермаркет, посетители которого прежде чем войти в торговый зал должны обязательно взять себе инвентарную тележку. В момент открытия магазина на входе имеется N свободных тележек – это начальное значение семафора. Каждый посетитель забирает одну из тележек (уменьшая тем самым количество оставшихся на 1) и проходит в торговый зал – это аналог операции down. При выходе посетитель возвращает тележку на место, увеличивая количество тележек на 1 – это аналог операции up. Теперь представим себе, что очередной посетитель обнаруживает, что свободных тележек нет – он вынужден **блокироваться** на входе в ожидании появления тележки. Когда один из посетителей, находящихся в торговом зале, покидает его, посетитель, ожидающий тележку, **разблокируется**, забирает тележку и проходит в зал. Таким образом, наш семафор в виде тележек позволяет находиться в торговом зале (аналоге критической секции) не более чем N посетителям одновременно. Положив N=1, получим реализацию взаимного исключения. Семафор, начальное (и максимальное) значение которого равно 1, называется двоичным семафором (т. к. имеет только 2 состояния: 0 и 1).

Семафоры – это низкоуровневые средства синхронизации, для корректной практической реализации которых необходимо наличие специальных, атомарных семафорных машинных команд.

4.2.3.2 Мониторы Хоара

Идея **монитора** была впервые сформулирована в 1974 г. Хоаром. В отличие от других средств, монитор представляет собой *языковую* конструкцию, т. е. Некоторое средство, предоставляемое языком программирования и поддерживаемое компилятором. Монитор представляет собой совокупность процедур и структур данных, объединенных в программный модуль специального типа. Постулируются три основных свойства монитора:

1. структуры данных, входящие в монитор, могут быть доступны только для процедур, входящих в этот монитор (таким образом, монитор представляет собой некоторый аналог объекта в объектно-ориентированных языках и реализует инкапсуляцию данных);
2. процесс «входит» в монитор путем вызова одной из его процедур;
3. в любой момент времени внутри монитора может находиться не более одного процесса. Если процесс пытается попасть в монитор, в котором уже находится другой процесс, он блокируется. Таким образом, чтобы защитить разделяемые структуры данных, из достаточно поместить внутрь монитора вместе с процедурами, представляющими критические секции для их обработки.

Монитор представляет собой конструкцию языка программирования и компилятору известно о том, что входящие в него процедуры и данные имеют особую семантику, поэтому первое условие может проверяться еще на этапе компиляции, кроме того, код для процедур монитора тоже может генерироваться особым образом, чтобы удовлетворялось третье условие. Поскольку организация взаимного исключения в данном случае возлагается на компилятор, количество программных ошибок, связанных с организацией взаимного исключения, сводится к минимуму.

4.2.4 Классические задачи синхронизации процессов

4.2.4.1 «Обедающие философы»

"Обедающие философы"

Пять философов собираются за круглым столом, перед каждым из них стоит блюдо со спагетти, и между каждыми двумя соседями лежит вилка. Каждый из философов некоторое время размышляет, затем берет две вилки (одну в правую руку, другую в левую) и ест спагетти, затем опять размышляет и так далее. Каждый из них ведет себя независимо от других, однако вилок запасено ровно столько, сколько философов, хотя для еды каждому из них нужно две. Таким образом, философы должны совместно использовать имеющиеся у них вилки (ресурсы). Задача состоит в том, чтобы найти алгоритм, который позволит философам организовать доступ к вилкам таким образом, чтобы каждый имел возможность насытиться, и никто не умер с голоду.

Рассмотрим простейшее решение, использующее семафоры. Когда один из философов хочет есть, он берет вилку слева от себя, если она в наличии, а затем - вилку справа от себя. Закончив есть, он возвращает обе вилки на свои места. Данный алгоритм может быть представлен следующим способом:

```
#define N 5                         /* число философов*/
void philosopher (int i)            /* i – номер философа от 0 до 4*/
{
while (TRUE)
      {
      think();                      /*философ думает*/
      take_fork(i);                 /*берет левую вилку*/
      take_fork((i+1)%N);           /*берет правую вилку*/
      eat();                        /*ест*/
      put_fork(i);                  /*кладет обратно левую вилку*/
      put_fork((i+1)%N);            /* кладет обратно правую вилку */
}
}
```

Функция **take_fork(i)** описывает поведение философа по захвату вилки: он ждет, пока указанная вилка не освободится, и забирает ее.

Данное решение может привести к тупиковой ситуации. Что произойдет, если все философы захотят есть в одно и то же время? Каждый из них получит доступ к своей левой вилке и будет находиться в состоянии ожидания второй вилки до бесконечности. Другим решением может быть алгоритм, который обеспечивает доступ к вилкам только четырем из пяти философов. Тогда всегда среди четырех философов по крайней мере один будет иметь доступ к двум вилкам. Данное решение не имеет тупиковой ситуации. Алгоритм решения может быть представлен следующим образом:

```
# define N 5              /* количество философов */
# define LEFT (i-1)%N     /* номер легого соседа для i-ого философа */
# define RIGHT (i+1)%N    /* номер правого соседа для i-ого философа*/
# define THINKING 0                 /* философ думает */
# define HUNGRY 1                   /* философ голоден */
# define EATING 2                   /* философ ест */
```

```
   typedef int semaphore;          /* определяем семафор */
   int state[N];              /* массив состояний каждого из философов */
   semaphore mutex=1;         /* семафор для критической секции */
   semaphore s[N];            /* по одному семафору на философа */

   void philosopher (int i)       /* i : номер философа от 0 до N-1 */
   {
      while (TRUE)                           /* бесконечный цикл */
      {
         think();               /* философ думает */
         take_forks(i);      /*философ берет обе вилки  или блокируется */
         eat();              /* философ ест */
         put_forks(i);       /* философ кладет обе вилки на стол */
      }
}

void take_forks(int i)      /* i : номер философа от 0 до N-1 */
{
      down(&mutex);               /* вход в критическую секцию */
      state[i] = HUNGRY;          /*записываем, что i-ый  философ голоден
*/
      test(i);                    /* попытка взять обе вилки */
      up(&mutex);                 /* выход из критической секции */
      down(&s[i]);                /* блокируемся, если вилок нет */
}

void put_forks(i)                 /* i : номер философа от 0 до N-1 */
{
      down(&mutex);               /* вход в критическую секцию */
      state[i] = THINKING;        /* философ закончил есть */
      test(LEFT);    /* проверить может ли левый сосед сейчас есть */
      test(RIGHT);   /* проверить может ли правый сосед сейчас есть*/
      up(&mutex);                 /* выход из критической секции */
}

void test(i)                      /* i : номер философа от 0 до N-1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING)
      {
         state[i] = EATING;
         up (&s[i]);
      }
}
```

4.2.4.2 Задача «читателей и писателей»

Другой классической задачей синхронизации доступа к ресурсам является проблема «читателей и писателей», иллюстрирующая широко распространенную модель совместного доступа к данным. Представьте себе ситуацию, например, в системе резервирования билетов, когда множество конкурирующих процессов хотят читать и обновлять одни и те же данные. Несколько процессов могут читать данные одновременно, но когда один процесс начинает записывать данные (обновлять базу данных проданных билетов), ни один другой процесс не должен иметь доступ к данным, даже для чтения. Вопрос, как спланировать работу такой системы? Одно из решений представлено ниже:

```
typedef int semaphore;           /* некий семафор */
semaphore mutex = 1;  /* контроль за доступом к «rc» (разделямый  ресурс)
*/
semaphore db = 1;     /* контроль за доступом к базе данных*/
int rc = 0;           /* кол-во процессов читающих или пишущих */

void reader (void)
{
      while (TRUE)                /* бесконечный цикл */
      {
            down (&mutex);   /* получить эксклюзивный доступ к «rc»*/
            rc = rc+1;       /* еще одним читателем больше */
                                    if (rc==1) down (&db);     /*  если
                                    это    первый    читатель,    нужно
                                    заблокировать  эксклюзивный  доступ
                                    к базе */
                  up(&mutex);       /*освободить ресурс rc */
            read_data_base();     /* доступ к данным */
                                    down(&mutex);    /*получить
                                    эксклюзивный доступ к «rc»*/
            rc = rc-1:            /* теперь одним читателем меньше */
                                    if (rc==0) up(&db);   /*если    это
                                    был       последний       читатель,
                                    разблокировать эксклюзивный доступ
                                  к базе данных */
            up(&mutex);           /*освободить разделяемый ресурс rc*/
            use_data_read();      /* некритическая секция */
      }
}

void writer (void)
{
      while(TRUE)                 /* бесконечный цикл */
      {
            think_up_data();      /* некритическая секция */
                                    down(&db); /*              получить
                                    эксклюзивный доступ к данным*/
            write_data_base();    /* записать данные */
            up(&db);              /* отдать эксклюзивный доступ */
      }
}
```

В этом примере, первый процесс, обратившийся к базе данных по чтению, осуществляет операцию DOWN над семафором **db**, тем самым блокируя эксклюзивный доступ к базе, который нужен для записи. Число процессов, осуществляющих чтение в данный момент, определяется переменной **rc** (обратите внимание! Т.к. переменная **rc** является разделяемым ресурсом – ее изменяют все процессы, обращающиеся к базе данных по чтению – то доступ к ней охраняется семафором **mutex**). Когда читающий процесс заканчивает свою работу, он уменьшает **rc** на единицу. Если он является последним читателем, он также совершает операцию UP над семафором **db**, тем самым разрешая заблокированному писателю, если такой имелся, получить эксклюзивный доступ к базе для записи.

Надо заметить, что приведенный алгоритм дает преимущество при доступе к базе данных процессам-читателям, т.к. процесс, ожидающий доступа по записи, будет ждать до тех пор, пока все читающие процессы не окончат работу, и если в это время появляется новый читающий процесс, он тоже беспрепятственно получит доступ. Это может привести к

неприятной ситуации в том случае, если в фазе, когда ресурс доступен по чтению, и имеется ожидающий процесс-писатель, будут появляться новые и новые читающие процессы. Чтобы этого избежать, можно модифицировать алгоритм таким образом, чтобы в случае, если имеется хотя бы один ожидающий процесс-писатель, новые процессы-читатели не получали доступа к ресурсу, а ожидали, когда процесс-писатель обновит данные. В этой ситуации процесс-писатель должен будет ожидать окончания работы с ресурсом только тех читателей, которые получили доступ раньше, чем он его запросил. Однако, обратная сторона данного решения в том, что оно несколько снижает производительность процессов-читателей, т.к. вынуждает их ждать в тот момент, когда ресурс не занят в эксклюзивном режиме.

4.2.4.3 Задача о «спящем парикмахере»

Рассмотрим парикмахерскую, в которой работает один парикмахер, имеется одно кресло для стрижки и несколько кресел в приемной для посетителей, ожидающих своей очереди. Если в парикмахерской нет посетителей, парикмахер засыпает прямо на своем рабочем месте. Появившийся посетитель должен его разбудить, в результате чего парикмахер приступает к работе. Если в процессе стрижки появляются новые посетители, они должны либо подождать своей очереди, либо покинуть парикмахерскую, если в приемной нет свободного кресла для ожидания. Задача состоит в том, чтобы корректно запрограммировать поведение парикмахера и посетителей.

Одно из возможных решений этой задачи представлено ниже. Процедура **barber()** описывает поведение парикмахера (она включает в себя бесконечный цикл – ожидание клиентов и стрижку). Процедура **customer()** описывает поведение посетителя. Несмотря на кажущуюся простоту задачи, понадобится целых 3 семафора: **customers** – подсчитывает количество посетителей, ожидающих в очереди, **barbers** – обозначает количество свободных парикмахеров (в случае одного парикмахера его значения либо 0, либо 1) и **mutex** – используется для синхронизации доступа к разделяемой переменной **waiting**. Переменная **waiting**, как и семафор **customers**, содержит количество посетителей, ожидающих в очереди, она используется в программе для того, чтобы иметь возможность проверить, имеется ли свободное кресло для ожидания, и при этом не заблокировать процесс, если кресла не окажется. Заметим, что как и в предыдущем примере, эта переменная является разделяемым ресурсом, и доступ к ней охраняется семафором **mutex**. Это необходимо, т.к. для обычной переменной, в отличие от семафора, чтение и последующее изменение не являются неделимой операцией.

```
#define CHAIRS 5
typedef int semaphore;           /* некий семафор */

semaphore customers = 0;    /* посетители, ожидающие в очереди */
semaphore barbers = 0;      /* парикмахеры, ожидающие посетителей */
                        semaphore mutex = 1;  /* контроль за доступом к
                        переменной waiting */
int waiting = 0;
void barber()
{
while (true) {
                      down(customers);      /* если customers == 0, т.е.
                      посетителей нет, то заблокируемся до появления
                      посетителя */
      down(&mutex);    /* получаем доступ к waiting */
      waiting = wating – 1; /* уменьшаем кол-во ожидающих клиентов */
      up(&barbers);         /* парикмахер готов к работе */
      up(&mutex);           /* освобождаем ресурс waiting */
      cut_hair();           /* процесс стрижки */
}
```

```
void customer()
{
      down(&mutex);          /* получаем доступ к waiting */
      if (waiting < CHAIRS)       /* есть место для ожидания */
      {
                                  waiting = waiting + 1;           /*
                                  увеличиваем кол-во ожидающих клиентов */
            up(&customers); /* если парикмахер спит, это его разбудит*/
            up(&mutex);                /* освобождаем ресурс waiting */
                                  down(barbers);   /*    если    парикмахер
                                  занят, переходим в состояние ожидания,
                                  иначе – занимаем парикмахера*/
            get_haircut();   /* занять место и перейти к стрижке */
      }
      else
      {
                        up(&mutex); /* нет свободного кресла для ожидания –
                        придется уйти */
      }
}
```

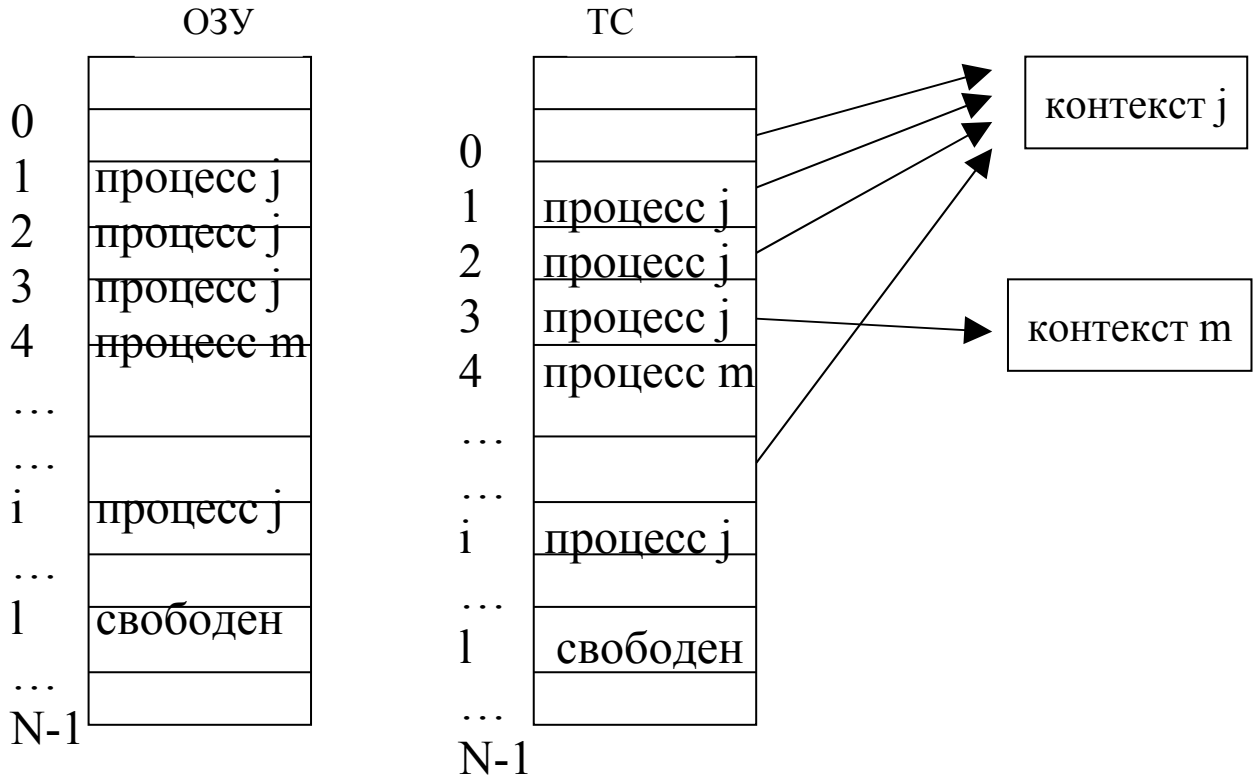
4.3 Управление оперативной памятью

Решение следующих задач:

- распределение физической памяти ОЗУ между процессами
- программная поддержка виртуальной памяти
- подкачка (свопинг)
- защита памяти

Конкретные алгоритмы зависят от свойств конкретной ЭВМ. Для модельной ЭВМ будем рассматривать страничную организацию ОЗУ. Пусть имеется ОЗУ, включающее до N физических страниц. Система команд машины позволяет адресовать до k страниц. Рассмотрим пример частичных действий модельно ОС по управлению ОП.

Операционная система формирует таблицу страниц (ТС):



Каждая строка ТС содержит информацию о статусе соответствующей физической страницы:

- свободна
- принадлежит j-му процессу (в этом случае в строке помещается ссылка на контекст соответствующего процесса)

Для каждого процесса, обрабатываемого в системе в данный момент времени (размещенного в БОП), ОС формирует программные структуры данных, в которых размещается информация контекста. Среди прочих значений в контексте размещается таблица страниц процесса (ТСП). По содержимому ТСП можно получить данные об используемых в процессе виртуальных страницах и их месторасположении. Под месторасположением считаем соответствие виртуальной страницы некоторой физической странице или указание координат места на ВЗУ, где размещена копия данной страницы. Соответственно поддержка решения задач управления ОП будет следующая.

При поступлении процесса в БОП заполняется ТСП. В начальный момент из описателей процесса, сформированных на этапе обработки в БВП выбирается список виртуальных страниц, который размещается в ТСП. Затем ОС анализирует содержимое ТС и «приписывает» виртуальным страницам их физические эквиваленты (при этом идет загрузка содержимого соответствующих виртуальных страниц из внешней памяти в физические страницы ОЗУ).

Для виртуальных страниц процесса, которым не были выделены физические страницы, в ТСП устанавливается признак отсутствия физической страницы (этот признак, также будет проставлен во все строки таблицы, соответствующие виртуальным страницам, не используемым процессом). Формируется содержимое таблицы «откаченных» страниц процесса ТОСП (указывается номер виртуальной страницы и ее месторасположение во внешней памяти).

Далее ОС из контекста данного процесса заполняет содержимое таблицы виртуальных страниц ТВС процессора и передает управление на начало выполнения процесса.

В рассмотренном примере затронуты элементы решения задач распределения физической памяти, поддержки использования аппарата виртуальной памяти, подкачки страниц. На самом деле в реальности полная логика действий существенно сложнее и окончательные настройки аппаратных средств (виртуальной памяти, защиты) есть вершина айсберга тех логически сложных действий, которые предшествуют чисто аппаратной реализации этих функций.

СВОППИНГ

ЗАЩИТА ПАМЯТИ

4.4 Планирование

Важной проблемой, на решение которой ориентированы многие компоненты современных ОС является проблема планирования предоставления тех или иных услуг или функций операционной системой. Традиционно, в состав задач планирования ОС могут входить следующие:

- планирование очереди процессов на начало обработки процессором;
- планирование распределения времени ЦП между обрабатываемыми в мультипрограммном режиме процессами;
- планирование порядка обработки заказов на обмен с ВУ;
- планирование порядка обработки прерываний;
- планирование использования ОЗУ (организация свопинга).

В целом, комплексное решение задач планирования в ОС определяет основные эксплуатационные качества каждой конкретной системы. Рассмотрим типовые задачи планирования и модельные решения этих задач.

4.4.1 Планирование очереди процессов на начало обработки ЦП

При планировании очереди процессов на начало обработки ЦП могут применятся как примитивные стратегии организации очереди FIFO, так и стратегии, учитывающие не только порядок поступления в очередь, но и объем ресурсов, продекламированных процессами для использования. В общем случае очередь процессов в БВП может предоставляться как объединение подочередей, где каждая подочередь включает в себя определенные классы процессов (например, такая классификация может строится на объеме запрашиваемых ресурсов и/или типе процесса). При этом возможно определение приоритета каждой из очередей (сначала рассматриваются непустые очереди с наименьшим приоритетом).

4.4.2 Планирование распределения времени работы ЦП между процессами

Здесь существует несколько проблем:

1. величина кванта времени работы ЦП, выделяемого выполняемому процессу.
2. стратегия выбора процесса, который будет выполняться ЦП из множества процессов, готовых к исполнению и размещенных в БОП.

4.4.3 Планирование очереди запросов на обмен

Определение порядка обработки запросов, разрешение проблем взаимосвязанных запросов на обмен.

4.4.4 Планирование порядка обработки прерываний

……

4.4.5 Типы операционных систем

Итак, комплексное решение проблем планирования однозначно определяет основные эксплуатационные характеристики и тип операционной системы.

4.4.5.1 Пакетная ОС

Рассмотри следующую модельную ситуацию. Пусть имеется ***пакет программ*** - некоторая совокупность программ, обладающих общим свойством – для выполнения каждой из программ необходимо значительное время работы ЦП. Необходимо обработать все программы пакета за минимальное время. Для этой цели используются специализированные ***пакетные ОС***. Для данных ОС не является важным порядок в котором будут выполнены программы пакета и время за которое была выполнена та или иная программа пакета. Критерием эффективности пакетной ОС является минимизация времени, затраченного на выполнение всего пакета за счет минимизации, в свою очередь, непроизводительной работы ЦП. Основной задачей системы планирования пакетной ОС является максимальная загрузка процессора мультипрограммным выполнением программ/процессов пользователей. В частности, должны быть минимизировано время работы ОС. Это достигается за счет стратегии планирования, основанной на переключении выполнения одной программы/процесса на другую только в одном из следующих случаев:

- завершение выполнения программы/процесса;
- возникновение при выполнении программы/процесса прерывания (например, обращение к ВУ);
- фиксация операционной системой факта зацикливания процесса.

Очевидно, что при подобной организации планирования соотношение времени работы процессора, затраченного на выполнение программ пользователей к времени, затраченному на выполнение функций ОС будет максимально.

4.4.5.2 Системы разделения времени

Другая модельная ситуация. В системе одновременно работает некоторое число пользователей, например, это может быть терминальный класс в котором проходят практические занятия и каждый пользователь работает со своей копией программы редактора или транслятора (или любой другой программой). Качеством работы пользователей в этой ситуации является создание иллюзии у пользователя, что он работает в системе один, т.е. время, в течении которого пользователь ожидает ответ системы на запрос должно быть минимально (запрос может быть нажатие кнопки исполнения на клавиатуре или отправка на выполнение отладочной программы и т.п.). Для работы системы в таких условиях используются ***операционные системы разделения времени***. Суть функционирования подобных систем заключается в следующем. В системе определено понятие ***квант времени ЦП –*** некоторый фиксированный ОС промежуток времени работы ЦП. Планирование в системах разделения времени осуществляется исходя из следующего. Каждому выполняющемуся в системе процессу выделяется квант времени ЦП, переключение выполнения на другой процесс осуществляется при:

- исчерпании процессом выделенного кванта времени;
- завершении выполнения программы/процесса;
- возникновении при выполнении программы/процесса прерывания (например, обращение к ВУ);
- фиксации операционной системой факта зацикливания процесса.

Мы можем видеть, что стратегия планирования пакетной ОС может быть сведена к стратегии ОС разделения времени (если считать размер кванта времени ЦП, выделяемого процессам при пакетной обработке равным бесконечности).

Характеристики конкретной системы разделения времени зависят от деталей стратегии распределения квантов ЦП, их величины, критериев выбора очередного процесса для выполнения.

4.4.5.3 ОС реального времени

Существует класс задач компьютерного управления теми или иными техническими объектами. Спецификой этих задач является реакция на события, возникающие при управлении в сроки, когда эта реакция имеет смысл (если Вам звонит кто-то по телефону, то имеет смысл поднять трубку до того времени как звонящий опустил свою трубку, поднятие трубки позднее – бессмысленно). Примеров подобных задач – множество, от сложных и, крайне ответственных, например, управление автопилотом самолета, до более прозаических и менее ответственных задач, например управление посудомоечной машиной. В общем случае, все подобные задачи имеет фиксированный набор некоторых событий, реакция на произвольное возникновение и обработка которых должна быть осуществлена за некоторое гарантированное время (возможно для каждого события это время может быть своим). ОС является ***системой реального времени*** если она при функционировании может обработать возникновение любого из данных событий (прерываний) за время, не превосходящее некоторое предельное значение. Системы реального времени являются специализированными системами в которых все функции планирования ориентированы на достижение поставленной цели.

4.5 Файловые системы

4.5.1 Основные свойства, функции, определения

Файловая система (ФС) - часть операционной системы, представляющая собой совокупность организованных наборов данных, хранящихся на внешних запоминающих устройствах, и программных средств, гарантирующих именованный доступ к этим данным и их защиту. Данные называются ***файлами***, их имена - ***именами файлов***.

Файловые системы можно классифицировать по степени персонификации доступа к содержимому файлов. Соответственно могут быть:

- однопользовательские файловые системы;
- многопользовательские файловые системы.

Однопользовательская **ФС** - система, в которой не регламентируется доступ к содержимому файлов от имени любого пользователя.

Многопользовательские файловые системы предусматривают работу только идентифицированных системой пользователей. Для многопользовательских файловых систем основным свойством является наличие защиты данных, содержащихся в файлах, от несанкционированного доступа.

Свойства файлов:

1. Файл представляет собой некую сущность, имеющую имя и позволяющую оперировать со своим содержимым через ссылку на имя файла.
2. Реальное местосторасположение данных файлов определяется файловой системой и в общем случае закрыто от пользователя.
3. Определен фиксированный программный интерфейс для работы с содержимым файла. Операционная система однозначно определяет набор функций, обеспечивающих обмен с файлом. Обычно, этот набор функций содержит следующие возможности по работе с файлами:
   1) ***Открытие файла***. Эта функция обеспечивает установление взаимосвязи между программой и хранящимся на внешнем носителе файлом. Это средство объявляет операционной системе тот факт, что с данным файлом будет работать тот или иной процесс. Операционная система, исходя из этой информации, может принять решения по изменению статуса файла (например, блокировать, разрешить или синхронизировать доступ к этому файлу со стороны других процессов и т.п). При открытии файла система формирует внутренние наборы данных, необходимые для работы с содержимым файла. С этими наборами данных ассоциируется понятие ***файлового дескриптора***.
   2) ***Закрытие файла***. Закрытие файла - информация операционной системе о том, что работа с файлом завершена. При этом меняется статус доступа к файлу со стороны процессов. Операция закрытия файла осуществляется двумя функциями:
   - закрыть и сохранить текущее содержимое файла;
   - уничтожить файл.
   3) ***Создание нового файла***. Функция создает новый файл. В некоторых ОС создание файла осуществляется по функции открытия файла.
   4) ***Чтение/запись***. Обычно обмен с файлами производится некоторыми блоками данных. С одной стороны, размеры этих блоков данных могут варьироваться программистом, с другой стороны, реальные физические ресурсы имеют блочную структура и, следовательно, определенный размер блока. Поэтому

эффективность обменов, а, следовательно, и эффективность работы всей ВС в целом, в данном случае зависит от квалификации программиста. Отметим, что в современных операционных системах заложены механизмы сглаживания подобного рода неэффективности.

5) ***Управление файловым указателем***. С каждым открытым файлом связано такое понятие, как ***файловый указатель***. Этот указатель в каждый момент времени показывает на следующий относительный адрес по файлу, с которым можно произвести обмен. После обмена с данным блоком указатель переносится на позицию через блок. Для организации работы с файлами необходимо уметь управлять этим указателем. В операционных системах определена функция, позволяющая произвольным образом перемещать указатель в пределах файла. Доступ к содержимому файла может быть прямыми и последовательным. Вообще говоря, файловый указатель есть некоторая переменная, доступная программе, которая создается при открытии файла.

4. Персонификация и защита данных. Персонификация – возможность системы «опознавать» конкретного пользователя и ассоциировать с ним его файлы. Защита доступа к содержимому файлов обычно включает в себя права на выполнение следующих действий:
   - чтение
   - запись
   - исполнение содержимого как процесс

Отметим, что персонификация и защита данных – это свойство всей ОС в целом.

4.5.2 Стратегии организации файловых систем

Рассмотрим некоторые типовые подходы к организации файловых систем.

4.5.2.1 Одноуровневая организация ФС с непрерывными сегментами

На внешнем запоминающем носителе выделяется некоторая непрерывная область. Данные размещаются в подряд идущих единицах этого носителя. В этой области в свою очередь выделяется подобласть для хранения информации о файлах, которая называется каталог. Каталог представляет собой таблицу, которая имеет три колонки: имя файла, координаты начала и конца файла, указанные в блоках. Имя файла в таблице должно быть уникальным (отсюда и термин – “одноуровневая”). При создании файла в эту таблицу добавляется строка с вышеперечисленными характеристиками. При уничтожении соответствующая строка удаляется из таблицы. Функция открытия уже существующего файла сводится к нахождению в каталоге имени файла, определении его начала и конца. Операции чтения/запись происходят почти без дополнительных обменов, так как при открытии файла мы получаем диапазон размещения данных (более того каталог можно хранить в оперативной памяти). Таким образом к несомненным достоинствам следует отнести простоту реализации и эффективность операций обмена.

Имя	Начало	Конец
cc	2352	2376
bp	2934	8024
delphi	15243	15678
…	…	…

Как отмечалось выше, особенностью этой организации является физическая непрерывность файла на внешнем носителе. Отсюда вытекает следующая проблема: при создании файла необходимо знать его максимальный размер, так как в случае необходимости добавления данных в файл может не оказаться достаточно места. В этом случае система может отказать в выполнении такой операции или может попытаться найти

новую непрерывную область достаточного размера, чтобы туда переписать данные. Последняя операция трудозатратна и приведет к внешней фрагментации, когда появятся незаполненные отрезки памяти между файлами. Выделение файлу места “заведомо достаточного” приведет к внутренней фрагментации. Разумеется можно осуществлять регулярную компрессию данных, но это операция также достаточно трудозатратна.

Такого рода организация может быть пригодна для организации однопользовательской файловой системы. Это объясняется тем, что

- при большом количестве пользователей очень быстро произойдет фрагментация, а постоянный запуск компрессии приведет к неэффективности системы
- ограниченность количества файлов в каталоге и необходимость уникального именования файлов делает многопользовательский режим крайне неудобным.

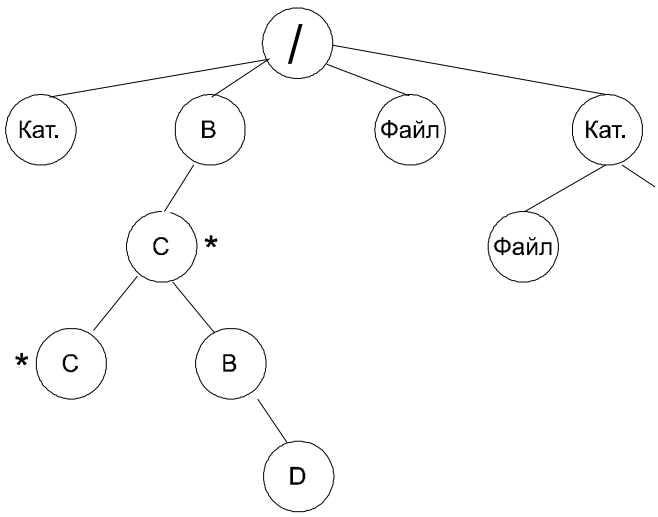
4.5.2.2 Файловая система с блочной организацией файлов

На пространстве внешней памяти выделяется непрерывная область данных, в которой размещается каталог. Вся оставшаяся внешняя память, выделенная для файловой системы, разбивается на блоки, удобные для обмена. Количество строк в каталоге совпадет с количеством этих блоков. Каждая строка таблицы соответствует i-му блоку файловой системы. Каждый файл занимает, как минимум, один блок памяти. Таблица разбивается на столбцы. Поле "имя" содержит имя файла, который занимает данный блок памяти. Поле "атрибуты" содержит различные подполя - имя пользователя, номера блоков, занимаемых файлом. Блоки, принадлежащие одному файлу, физически могут располагаться в произвольном порядке. Такой способ организации файловой системы решает проблему лимитирования размера файла. При создании нового файла его размер будет всегда равным 1 блоку, если при записи в файл его фактический размер превышает размер одного блока, то в таблицу записывается новая строка с именем этого файла и соответствующими атрибутами — это означает, что к файлу присоединяется еще один блок и так далее. Незаполненные строки таблицы образуют список свободной памяти. При такой организации исчезает проблема внешней фрагментации, но актуальна проблема внутренней фрагментации. Так как даже если реальный размер файла равен 1 байту, он все равно займет целый блок памяти. Уменьшение размера блока уменьшает внутреннюю фрагментацию, но увеличивает размер таблицы. Последнее усложняет с ней работу. К примеру, при открытии файла необходимо просмотреть всю таблицу, чтобы определить все блоки файла. Если таблица не хранится в оперативной памяти, то увеличивается количество необходимых обменов.

Приведенная организации файловой системы является одноуровневой в рамках одного пользователя, т.е. все файлы связаны в группы по принадлежности к одному пользователю. Таким образом уникальность имен требуется только среди файлов одного пользователя.

4.5.2.3 Иерархические файловые системы

За основу логической организации такой файловой системы берется дерево. В корне дерева находится, так называемый, корень файловой системы - каталог нулевого уровня. В этом каталоге могут находиться либо файлы пользователей, либо каталоги первого уровня. Каталоги первого и следующих уровней организуются по аналогичному принципу. Файлы пользователя в этом дереве представляются листьями. Пустой каталог также может быть листом. Таким образом образуется древовидная структура файловой системы, где в узлах находятся каталоги, а листьями являются либо файлы, либо пустые каталоги.



Остановимся на правилах именования в иерархической файловой системе. В

данном случае используется механизм, основанный на понятии имени файла (name) и полного имени файла (path_name). Полное имя файла – это путь от корневого каталога до листа (такой путь всегда будет уникальным). Существует также относительное именование, т.е. когда нет необходимости указания полного пути при работе с файлами. Это происходит в случае, когда программа вызывает файл и подразумевается, что он находится в том же каталоге, что и программа. В данном случае появляется понятие текущего каталога, т. е. каталога, на работу с которым настроена файловая система в данный момент времени. В рамках одного каталога имена файлов одного уровня должны быть разными.

Согласно этой иерархии, с каждым из файлов можно ассоциировать атрибуты, связанные с правами доступа. Правами доступа могут обладать как файлы, так и каталоги.

Структура этой системы удобна для организации многопользовательской работы, за счет приведенного выше способа именования и возможности удобного наращивания ФС .

4.6 Управление внешними устройствами

Управление устройствами и организация обмена.
Обработка прерываний, учет приоритетов прерываний, буферизация обмена.

5 ОС Unix: Файловая система.

5.1 Особенности, характеристики Unix:

- в основном архитектура системы построена на использовании и организации понятий файл и процесс;
- открытость, прозрачность системы;
- унификация основных интерфейсов;
- иерархическая файловая система;
- развитый аппарат управления и взаимодействия процессов;
- «переносимость» системы;
- развитая внутренняя оптимизация работы компонентов системы.

5.2 Организация файловой системы Unix. Пользовательский аспект.

Файловая система операционной системы **UNIX** является примером многопользовательской иерархической файловой системой с трехуровневой организацией прав доступа к содержимому файлов .

Файл Unix – это специальным образом именованный набор данных, размещенный в файловой системе.

ОС Unix трактует понятие файла шире традиционного. В частности, в системе в качестве файла рассматриваются :

- ***обычный файл*** (regular file) – традиционный тип файла, содержащий данные пользователя. Интерпретация содержимого файла производится программой, обрабатывающей файл.
- ***каталог*** (directory) – специальный файл, обеспечивающий иерархическую организацию файловой системы. С каталогом ассоциируются все файлы, которые принадлежат данному каталогу.
- ***специальный файл устройств*** (special device file) – система позволяет ассоциировать внешние устройства с драйверами и предоставляет доступ к внешним устройствам, согласно общим интерфейсам работы с файлами.
- ***именованный канал*** (named pipe) – специальная разновидность файлов, позволяющая организовывать передачу данных между взаимодействующими процессами;
- ***ссылка*** (link) – позволяет создавать дополнительные ссылки к содержимому файла из различных точек файловой системы;
- ***сокет*** (socket) – средство взаимодействия процессов в пределах сети ЭВМ.

Права доступа к содержимому файлов в системе жестко связаны с организацией пользователей системы. С точки зрения организации прав доступа к содержимому файлов рассматриваются следующие категории пользователей:

- ***пользователь*** – владелец файла;
- ***группа*** – категория, к которой принадлежит пользователь – владелец файла, за исключением самого этого пользователя;
- ***все пользователи системы*** - все остальные пользователи системы за исключением первых двух категорий пользователей.

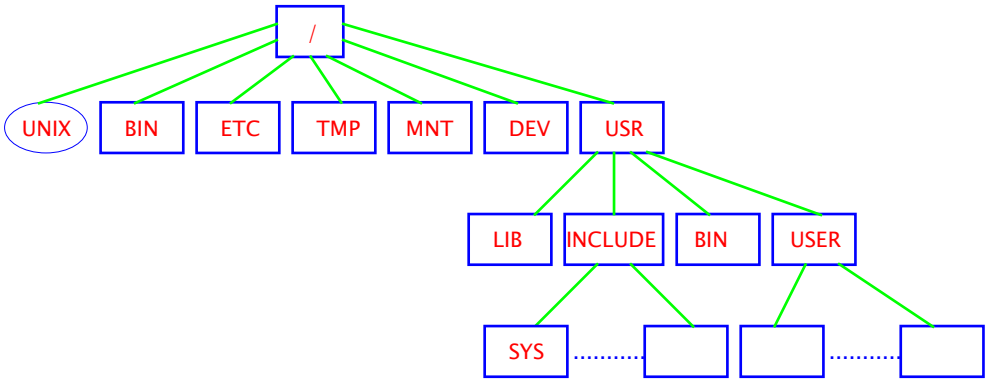
Для каждой из перечисленных выше категорий определены ***права*** на выполнение следующих действий:

- чтение;
- запись;
- исполнение.

Интерпретация этих прав зависит от типа файла. Так для обычных файлов это традиционные права на чтение, запись данных файла и исполнение содержимого файла в качестве процесса. Интерпретация прав доступа для других типов файлов может различаться. Например, для файлов каталогов это:

- право на чтение каталога – получение списка имен файлов;
- право на исполнение каталога – получение дополнительной информации о файлах (т.е. тогда, когда требуется информация, большая чем имя файла), право на использование каталога в качестве текущего, возможность использования имени каталога внутри имени файла;
- право на запись – возможность создания, переименования и удаления файла в каталоге.

Все UNIX-системы имеют соглашения о логической структуре каталогов, расположенных в корне файловой системы. Это упрощает работу операционной системы, ее обслуживание и переносимость. Эти соглашения используются при работе почтовой системы, системы печати и т.д.
Приведем описание содержимого основных каталогов:



Корневой каталог /
является основой любой файловой системы ОС UNIX. Все остальные файлы и каталоги располагаются в рамках структуры, порожденной корневым каталогом, независимо от их физического положения на диске.
/unix - файл загрузки ядра ОС.
/bin - файлы, реализующие общедоступные команды системы.
/etc - в этом каталоге находятся файлы, определяющие настройки системы (в частности, файл passwd), а также команды, необходимые для управления содержимым подобных специальных файлов.
/tmp - каталог для хранения временных системных файлов. При перезагрузке системы не гарантируется сохранение его содержимого. Обычно этот каталог открыт на запись для всех пользователей системы.
/mnt - каталог, к которому осуществляется монтирование дополнительных физических файловых систем для получения единого дерева логической файловой системы. Заметим, что это лишь соглашение, в общем случае можно примонтировать к любому каталогу.
/dev - каталог содержит специальные файлы устройств, с которыми ассоциированы драйверы устройств. Каждый из файлов имеет ссылку на соответствующий драйвер и указание типа устройства (блок- или байт-ориентированные). Этот каталог может содержать несколько подкаталогов, группирующих специальные файлы по типам. Таким образом, имеется возможность легко добавлять и удалять новые устройства в систему.
/lib - здесь находятся библиотечные файлы языка Си и других языков программирования.
/usr - размещается вся информация, связанная с обеспечением работы пользователей. Здесь также имеется подкаталог, содержащий часть библиотечных файлов (**/usr/lib**), подкаталог

/usr/users (или **/usr/home**), который становится текущим при входе пользователя в систему, подкаталог, где находятся дополнительные команды (**/usr/bin**), подкаталог, содержащий файлы заголовков (**/usr/include**), в котором, в свою очередь подкаталог, содержащий include-файлы, характеризующие работу системы (например, signal.h - интерпретация сигналов).

5.3 Внутренняя организация файловой системы

5.3.1 Модель версии system V

Файловая система Unix может занимать раздел диска (partition). Количество разделов на каждом диске, их размеры определяются при предварительной подготовке устройства (разметка). Unix рассматривает разделы, как отдельные, независимые устройства.

Структура файловой системы:

Суперблок	Область индексных дескрипторов	Блоки файлов

Суперблок файловой системы
содержит оперативную информацию о текущем каталоге файловой системы, а также данные о параметрах настройки, в частности:

- размер логического блока (512б, 1024б, 2048б);
- размер файловой системы в логических блоках (включая суперблок);
- максимальное количество индексных дескрипторов (определяет размер области индексных дескрипторов);
- число свободных блоков;
- число свободных индексных дескрипторов;
- специальные флаги;
- массив номеров свободных блоков;
- массив номеров свободных индексных дескрипторов;
- и др.

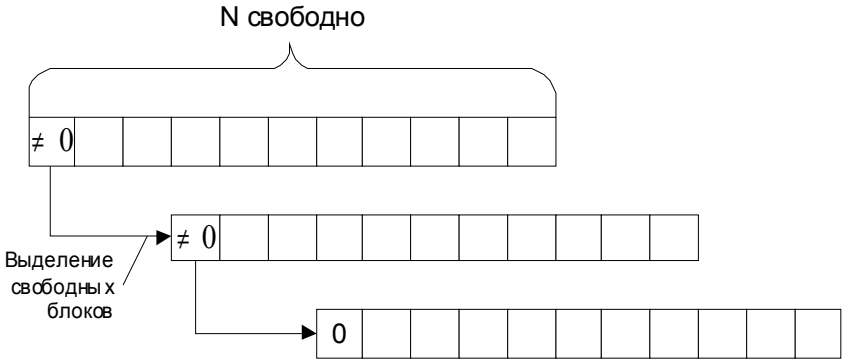
Область (пространство) индексных дескрипторов.
Индексный дескриптор – это специальная структура данных файловой системы, которая ставится во взаимно однозначное соответствие с каждым файлом. Размер пространства индексных дескрипторов определяется параметром генерации файловой системы по количеству индексных дескрипторов, которые указаны в суперблоке.

Следующее пространство файловой системы - это **блоки файлов**. Это пространство на системном устройстве, в котором размещается вся информация, хранящаяся в файлах и о файлах, которая не поместилась в предыдущие блоки файловой системы.

Рассмотрим понятия, связанные с ключевыми атрибутами файловой системы и базовые алгоритмы работы с ними.

Работа с массивами номеров свободных блоков

В суперблоке файловой системы размещается ***массив номеров свободных блоков,*** этот массив является началом полного списка содержащего номера всех свободных блоков файловой системы. Пусть $N_{свободд.}$ размер массива (зависит от размера блока).



Нулевой элемент массива содержит признак продолжения списка (непосредственно номера блоков размещаются с первого элемента до конца массива). Если массив полностью заполнен, то в этом случае нулевой элемент массива содержит номер блока в котором размещается продолжение списка свободных блоков и т.д. В противном случае нулевой элемент равен нулю. Оперативный доступ к списку осуществляется посредством использования массива в суперблоке.

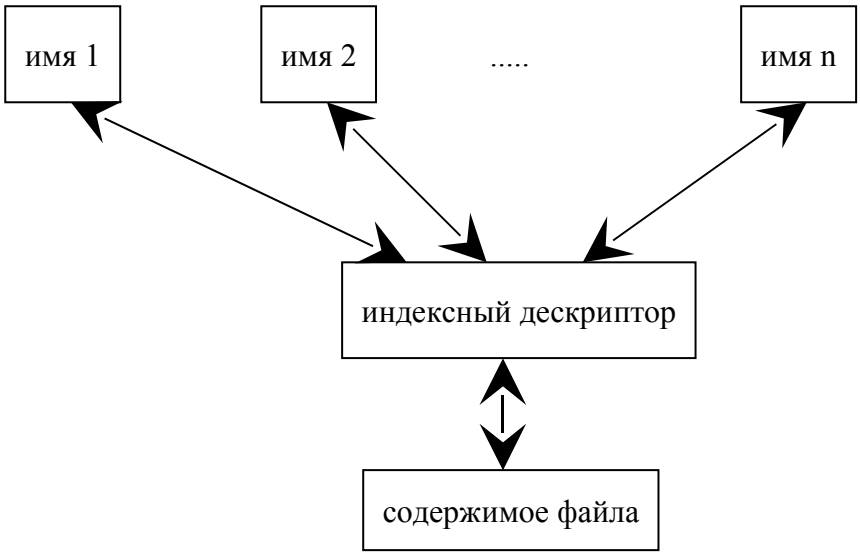
Работа с массивом свободных индексных дескрипторов

Массив номеров свободных индексных дескрипторов содержит оперативный набор номеров свободных индексных дескрипторов. Размер массива - $N_{индекс}$. При освобождении индексного дескриптора, если есть свободное место в массиве, то номер освободившегося индексного дескриптора записывается в соответствующий элемент массива. Если свободного места в массиве нет, то этот номер «забывается». При запросе нового индексного дескриптора осуществляется поиск в массиве, если массив не пустой, то все в порядке, если массив пустой – происходит операция обновления его содержимого (происходит просмотр области индексных дескрипторов и занесение в массив обнаруженных свободных). Т.е. массив свободных индексных дескрипторов – это своеобразный буфер.

Индексные дескрипторы

Индексный дескриптор (ИД) – описатель файла, содержит все необходимые для работы с файлом служебные атрибуты.

Через ИД осуществляется доступ к содержимому файлов. Любое имя файла в системе ассоциировано с ИД, но это соответствие неоднозначно. Т.е. ИД может соответствовать произвольное количество имен.



Структура индексного дескриптора:

- тип файла, права, атрибуты выполнения (если = 0, то ИД свободен);
- число имен, которые ассоциированы с данным ИД;
- идентификаторы владельца-пользователя, владельца-группы;
- размер файла в байтах;
- время последнего доступа к файлу;
- время последней модификации содержимого файла;
- время последней модификации ИД (за исключением времени доступа и времени модификации файла)
- массив номеров блоков файла.

Адресация блоков файла.

Интересным представляется организация адресации блоков файлов, принятая в данной модели файловой системы. Рассмотрим подробнее.
Для простоты изложения будем считать, что размер блока равен 512 байт.

Размещение данных файла задается списком его блоков. Это снимает проблемы непрерывных файловых систем, т.е. систем, где блоки файла располагаются последовательно. Таким образом реально блоки файла могут быть разбросаны по диску, но логически они образуют цепочку, содержащую весь набор данных. Ключом, задающим подобное расположение служит массив номеров блоков файла, содержащий список из 13 номеров блоков на диске, хранящихся в **ИД**. Первые десять указывают на десять блоков некоторого файла. Если файл занимает более 10 блоков, то 11 элемент указывает на косвенный блок, содержащий до 128 адресов дополнительных блоков файла (это еще 70656 байт). Большие файлы используют 12 элемент, который указывает на блок, содержащий 128 указателей на блоки, каждый из которых содержит по 128 адресов блоков файла. Еще в больших файлах аналогично используется 13 элемент. Трехкратная косвенная адресация позволяет создавать файлы длиной (10+128+128*128+128*128*128)*512 байт. Таким образом, если файл меньше 512 байт, то необходимо одно обращение к диску, если длина файла находится в пределах 512-70565 байт, то - два и так далее. Приведенный способ адресации позволяет иметь прямой и быстрый доступ к файлам. Эта возможность также усиливается кэшированием диска, позволяющим хранить в памяти наиболее используемые блоки . Важно отметить, что при ***открытии файла*** соответствующий **ИД** считывается в память и системе становятся доступны все номера блоков данного файла. Кроме того, для одного и того же файла, открываемого несколько раз, в памяти находится только один **ИД.** Система фиксирует число открытий данного файла и, когда этот счетчик обнуляется, резидентный образ **ИД** переписывается на диск. Если при этом изменений в файле не было и не модифицировался **ИД,** то запись не выполняется. Указанные особенности существенно влияют на эффективность файловой системы.

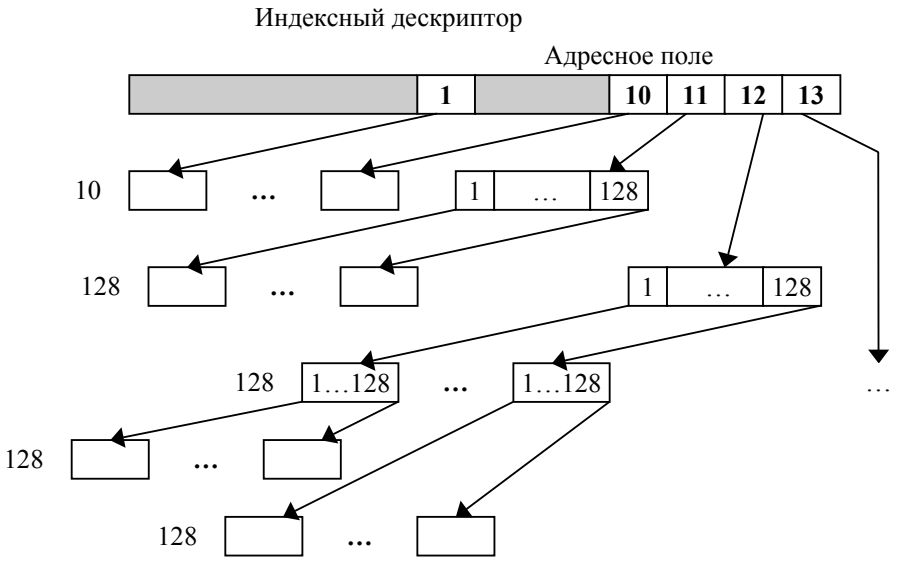
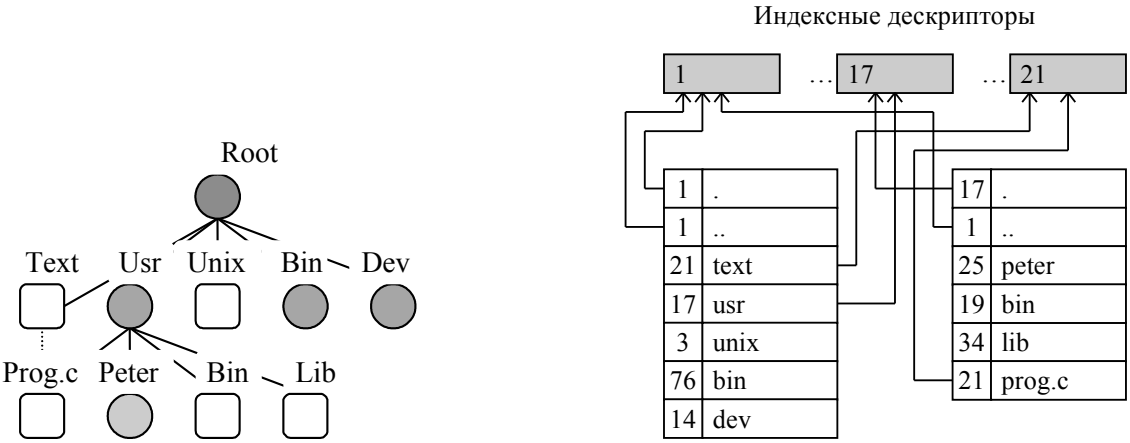


Рис.5 Адресация данных файла.

Файл каталог.

Содержимое файла – таблица. 1-е поле – это номер индексного дескриптора (ИД), которому соответствует имя Name из второго поля. Размеры полей в общем случае могут быть различные. Например размер поля ИД – 2 байта (ограничение числа ИД в файловой системе 65535), размер поля Name – 14 байт (соответственно ограничение на длину имени). В Unix две первые строки любого каталога имеют фиксированное содержание: имя «•» - ссылка на самого себя, имя «••» - ссылка на родительский каталог.

Видно, что при такой реализации имя файла “отделено” от других его атрибутов. Это позволяет, в частности, один и тот же файл внести в несколько каталогов. При этом, как отмечалось выше, данный файл может иметь разные имена в разных каталогах, но ссылаться они будут на один и тот же **ИД**, который является ключом для доступа к данным файла. При обсуждении понятия **ИД** говорилось, что каждая новая ссылка к **ИД** отмечается в специальном поле. Рассмотрим пример, иллюстрирующий связь файлов с каталогами.



Установление связей

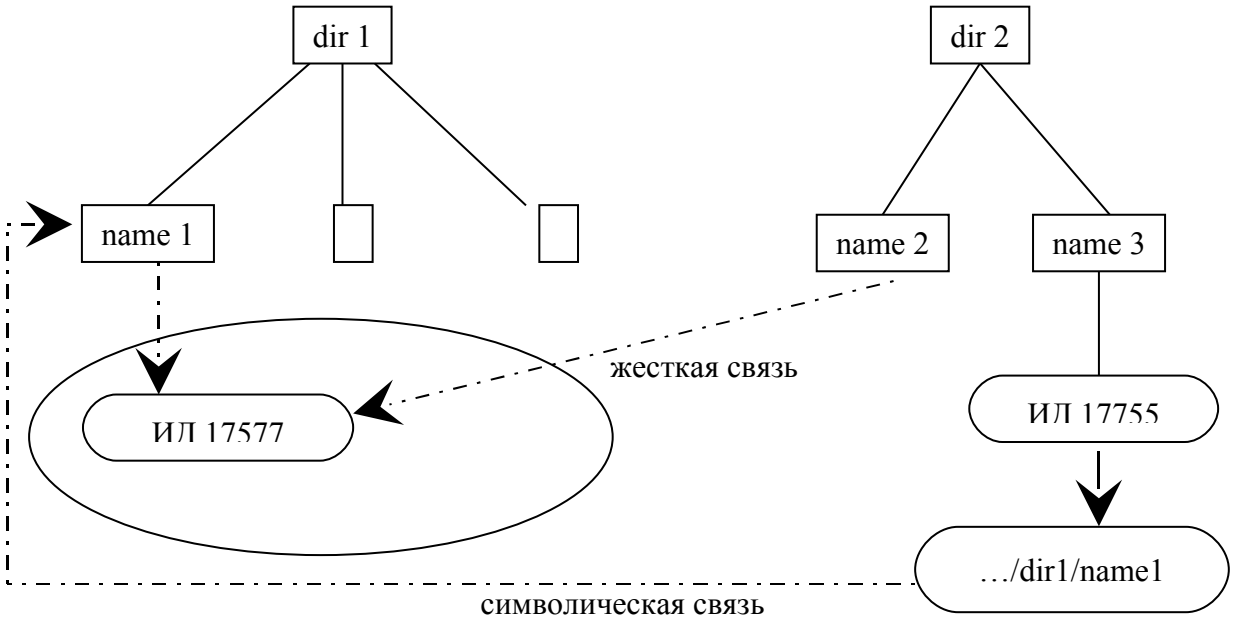
Древовидность файловой системы Unix нарушается возможностью установления ссылок на одни и те же индексные дескрипторы из различных каталогов. Это может быть достигнуто за счет использования средств установления дополнительных связей. Существует две разновидности этой операции.

Установление ***жесткой связи*** - образование дополнительного имени, ассоциированного с индексным дескриптором. Для этого используется команда:

ln …dir1/name1 …dir2/name2 – (дли индексного дескриптора, с которым ассоциировано имя **name1** добавляется еще одно имя – **name2**). Все имена, ассоциированные таким образом с индексным дескриптором равноправны. При этом увеличивается значение поля индексного дескриптора **число имен, которые ассоциированы с данным ИД.** Нельзя устанавливать ***жесткую связь*** для файлов-каталогов.

Установление ***символической связи*** - косвенная адресация на существующее имя файла. Для этих целей используется команда:

ln –s …dir1/name1 …dir2/name3 – в результате образуется специальный ***файл - ссылка***



Система работает с ним особым образом (например, при просмотре осуществляется просмотр не файла-ссылки, а файла на который он ссылается).

Достоинства/недостатки данной модели файловой системы

(+) Оптимизация в работе со списками номеров свободных индексных дескрипторов и блоков.

(+) Организация косвенной адресации блоков файлов, позволяющая использовать эффективный доступ к значительному количеству блоков файла.

(–) Концентрация важной информации в суперблоке - ключевая информация сконцентрирована в суперблоке файловой системы, физическая потеря содержимого суперблока может приводит к значительным проблемам, касающимся целостности файловой системы.

(–) Проблема надежности (много ссылочных структур, возможна потеря данных при сбоях).

(–) Фрагментация файла по диску – т.е. при достаточно больших размерах файла его блоки могут произвольным образом размещаться на физическом МД? Что может приводить к выполнению значительного числа механических операций перемещения головок устройства при чтении/записи данных файла.

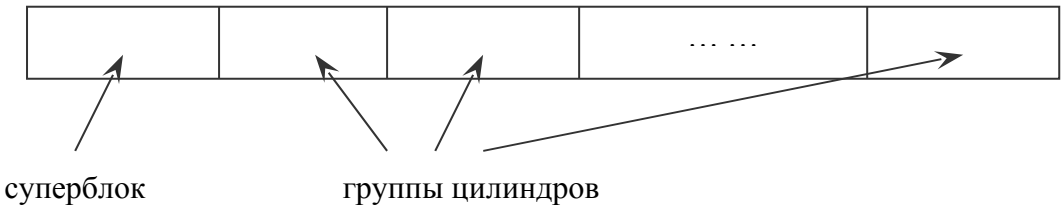
(-) Организация каталога накладывает ограничения на возможную длину имени файла (14 символов).

5.3.2 Модель версии FFS BSD

В Unix 4.2 BSD разработана модель организации файловой системы, которая получила название Fast File System - ffs (быстрая файловая система). Основной идеей данной модели файловой системы является кластеризация дискового пространства файловой системы, с

целью минимизации времени чтения/записи файла, а также уменьшения объёма неиспользуемого пространства внутри выделенных блоков.

Суть кластеризации заключается в следующем. Дисковое пространство, так же, как и в модели s5fs имеет суперблок, в котором размещена ключевая информация файловой системы (структура суперблоков s5fs и ffs, в общем случае, логически идентична), далее, дисковое пространство разделено на области одинакового размера, называемые группами цилиндров. Далее, стратегия функционирования файловой системы такова, что она старается разместить содержимое файлов (блоки файлов) в пределах одной группы цилиндров, при этом стараясь располагать файлы в той же группе цилиндров, что и каталог в котором они расположены.

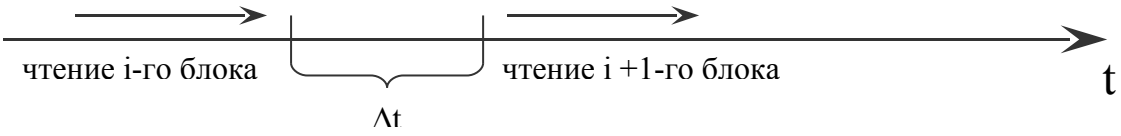


Группа цилиндров:

- копия суперблока;
- информация о свободных блоках и индексных дескрипторах;
- массив индексных дескрипторов (ИД);
- блоки файлов.

Стратегии размещения:

- новый каталог помещается в группу цилиндров, число свободных индексных дескрипторов в которой больше среднего значения во всей файловой системе в данный момент времени, а также имеющей минимальное число дескрипторов каталогов в себе;
- для обеспечения равномерности использования блоков данных файл разбивается на несколько частей, при этом первая часть файла располагается в той же группе цилиндров, что и его дескриптор, при размещении последующих частей используется группа цилиндров, в которой число свободных блоков превышает среднее значение. Длина первой части выбирается таким образом, чтобы она адресовалось непосредственно индексным дескриптором (т.е. не «косвенно»), остальные части разбиваются фиксированным образом, например по 1 мегабайту;
- последовательные блоки файлов размещаются исходя из оптимизации физического доступа (см. ниже)



Δt – технологический промежуток времени, который затрачивает система на передачу и прием устройством МД команды на чтение очередного блока. За это время диск проворачивается и головки обмена «пропускают» начало очередного блока, поэтому если мы будем читать следующий блок, то головка

будет вынуждена ожидать полного поворота диска на начало блока. В связи с этим эффективнее читать не последовательные блоки (в этом случае нужно ожидать полного поворота диска), а блоки, размещенные на диске через один, два... (смещение определяется поворотом диска за время Δt).

Внутренняя организация блоков

Обмен происходит блоками. Блоки могут быть достаточно большого размера (до 64 Кб). В системе может быть принято разбиение блока на равные фрагменты (на 2, 4, 8). То есть все пространство разделяется на «маленькие блоки» - фрагменты. Фрагменты группируются по 2, 4 или 8 в блоки (т.е. если фрагмент содержит 512 байт, то блок может быть размера 1024, 2048, 4096). Блоки выровнены по кратности.

Блоки	0				1				…	N			
Фрагменты	0	1	2	3	4	5	6	7	…				
Маска	1	0	0	0	0	1	1	1					

При этом блоком в этой системе может называться только «выровненный» до размера кратности набор фрагментов. Т.е. при кратности 4 (см. рисунок выше), фрагменты 0 – 3 – входят в один блок, а фрагменты 1 – 4 нет.
Для хранения информации о свободных фрагментах используется битовая маска: каждому фрагменту на диске соответствует ровно 1 бит в этой маске (этот механизм упрощает алгоритм поиска свободных фрагментов и уменьшает «фрагментацию» свободного пространства).
Формат индексного дескриптора аналогичен, используемому в s5fs - в нём в качестве элементов по-прежнему используются блоки, а не фрагменты, но при размещении информации в файлах используется следующее простое правило: все блоки указанные в индексном дескрипторе, кроме последнего, должны использоваться только целиком; блок может использоваться для нескольких файлов только при хранении их последних байт, не занимающих всех фрагментов полного блока (см. рисунок ниже). Т.о. для хранения информации об использовании последнего блока недостаточно только размера файла, хранимого в дескрипторе System 5, необходимо также хранить информацию, об используемых фрагментах в этом блоке.

Выделение пространства для файла происходит только в момент когда процесс выполняет системный вызов write (см. рисунок ниже). Операционная система при этом руководствуется следующим алгоритмом:

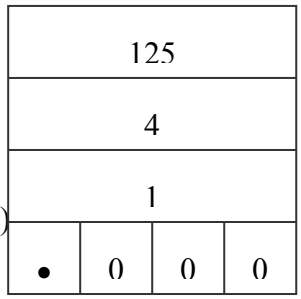
1. Если в уже выделенном файлу блоке есть достаточно места, то новые данные помещаются в это свободное пространство.
2. Если последний блок файла использует все фрагменты (т.е. это полный блок) и свободного в нём места не достаточно для записи новых данных, то частью новых данных заполняется всё свободное место. Если остаток данных превышает по размеру один полный блок, то новый выделяется полный блок и записываются данные в этот полный блок. Процесс повторяется до тех пор, пока остаток не окажется меньше чем полный блок. В этом случае ищется блок с необходимыми по размеру фрагментами или выделятся новый полный блок. Остаток данных записывается в этот блок.
3. Файл содержит один или более фрагмент (они естественным образом содержатся в одном блоке) и последний фрагмент недостаточен для записи новых данных. Если размер новых данных в сумме с размером данных, хранимых в неполном блоке, превышает размер полного блока, то выделяется новый полный блок. Содержимое старого неполного блока копируется в начало выделенного блока и остаток

заполняется новыми данными. Процесс далее повторяется, как указано в пункте 2 выше. В противном случае (если размер новых данных в сумме с размером данных, хранимых в неполном блоке, не превышает размер полного блока) ищется блок с необходимыми по размеру фрагментами или выделятся новый полный блок. Остаток данных записывается в этот блок.

Структура каталога

Поддержка длинных имен файлов.
Любая запись содержит:

- номер индексного дескриптора;
- длина записи в каталоге;
- длина имени файла;
- имя файла (дополненное до кратности слова)



При удалении имени файла принадлежащая ему запись присоединяется к предыдущей (при этом увеличивается значение поля «длина записи»), при этом если освобождается последняя запись в выделенном фрагменте, то он помечается как свободный и удаляется из ИД директории.

Символические ссылки

Стандартная файловая система, позволяла использовать только жёсткие ссылки, при этом в каталоге помещалась ссылка на файл. Допускалось иметь несколько записей в каталоге, ссылающихся на один и тот же ИД. Это вносило ограничение на создание ссылок между разными файловыми системами. В s5fs был добавлен файл нового типа – символическая ссылка, при этом его содержимым был путь (относительный или абсолютный) к файлу, на который эта ссылка установлена. Для обычных программ это изменение незаметно – интерпретацией символических ссылок по-прежнему занимается операционная система.

Другие изменения

В s5fs были добавлены ещё некоторые механизмы, такие как блокировка файлов, поддержка переименования файлов (системный вызов rename), поддержка квотирования дискового пространства.

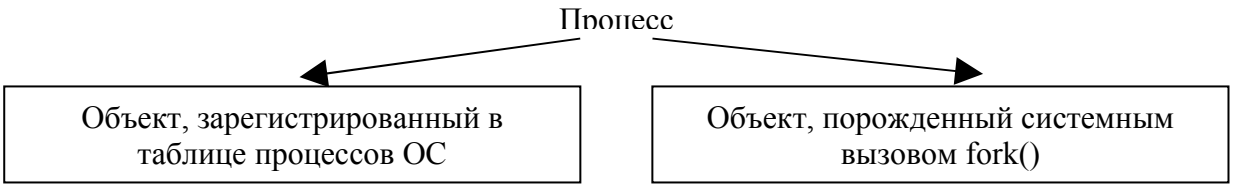
6 ОС Unix: процессы, взаимодействие процессов

6.1 Базовые свойства процессов

6.1.1 Определение процесса. Контекст

В любой системе, оперирующей понятием процесс, существует системно-ориентированное определение процесса (определение, учитывающее конкретные особенности данной ОС).

С точки зрения Unix системно-ориентированное определение процесса:



Рассмотрим данные определения процесса Unix.

Первое определение процесса Unix.

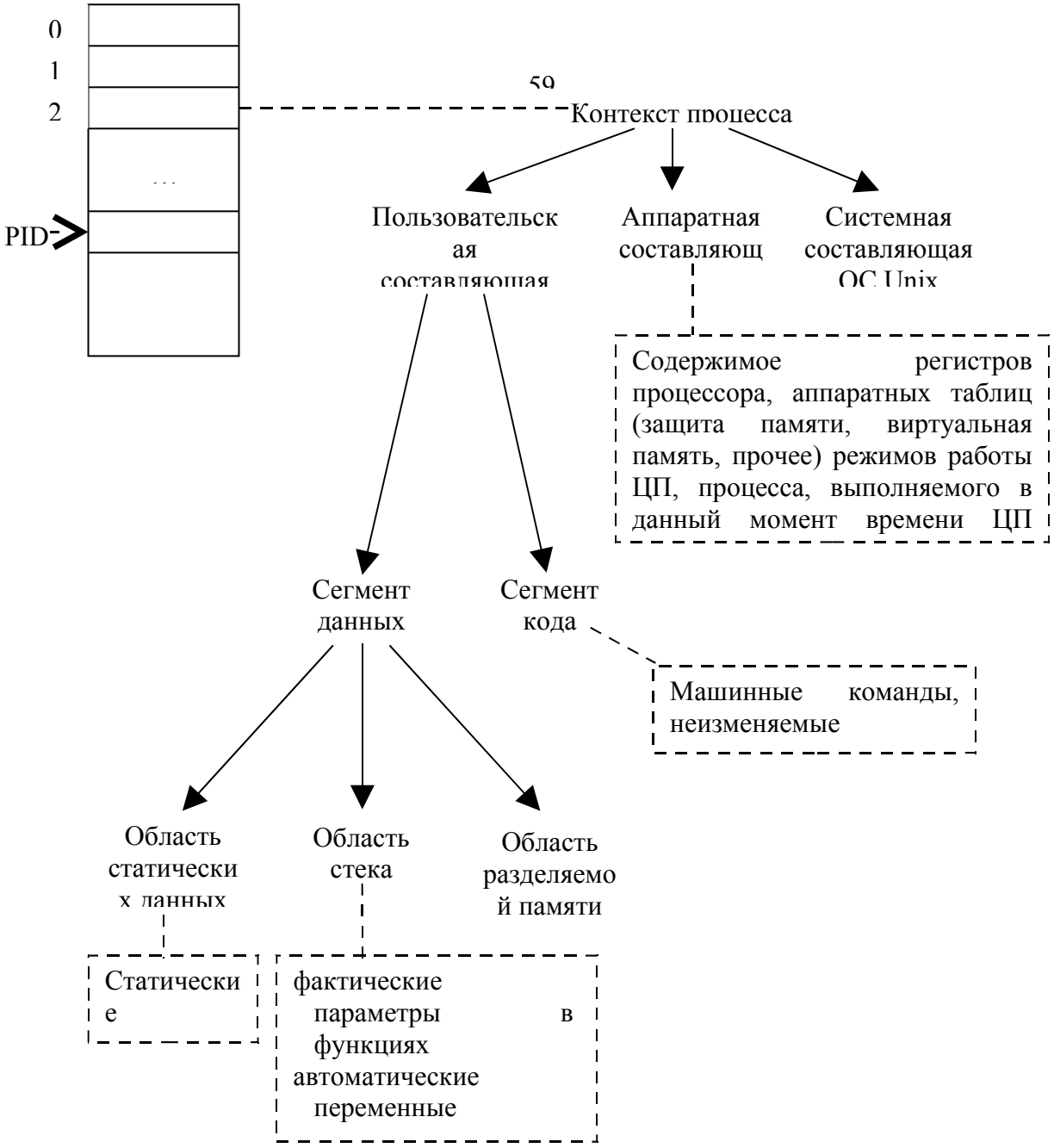
Процесс в ОС Unix – объект (не надо путать с объектом ООП!!!), зарегистрированный в таблице процессов Unix.

Таблица процессов

Каждый процесс характеризуется уникальным именем – ***идентификатором процесса*** (PID). PID – целое число от 0 до некоторого предельного значения, определяющего максимальное число процессов, существующих в системе одновременно.

Будем использовать термины 0й процесс, 1й процесс, 125й процесс, это означает, что речь идет о процессах с PID = 0, 1, 125. 0й процесс в системе ассоциируется с работой ядра Unix. С точки зрения организации данных PID – номер строки в таблице, в которой размещена запись о процессе.

Контекст процесса



Содержимое записи таблицы процессов позволяет получить ***контекст процесса*** (часть данных контекста размещается непосредственно в записи таблицы процессов, на оставшуюся часть контекста имеются прямые или косвенные ссылки, также размещенные в записи таблицы процессов).

С точки зрения логической структуры контекст процесса Unix состоит из:

- ***пользовательской составляющей*** или ***тела процесса*** (иногда используется пользовательский контекст)
- ***аппаратной составляющей*** (иногда используется аппаратный контекст)
- ***системной составляющей*** ОС Unix (иногда – системный контекст)

Иногда два последних компонента объединяют, в этом случае используется термин ***общесистемная составляющая контекста***.

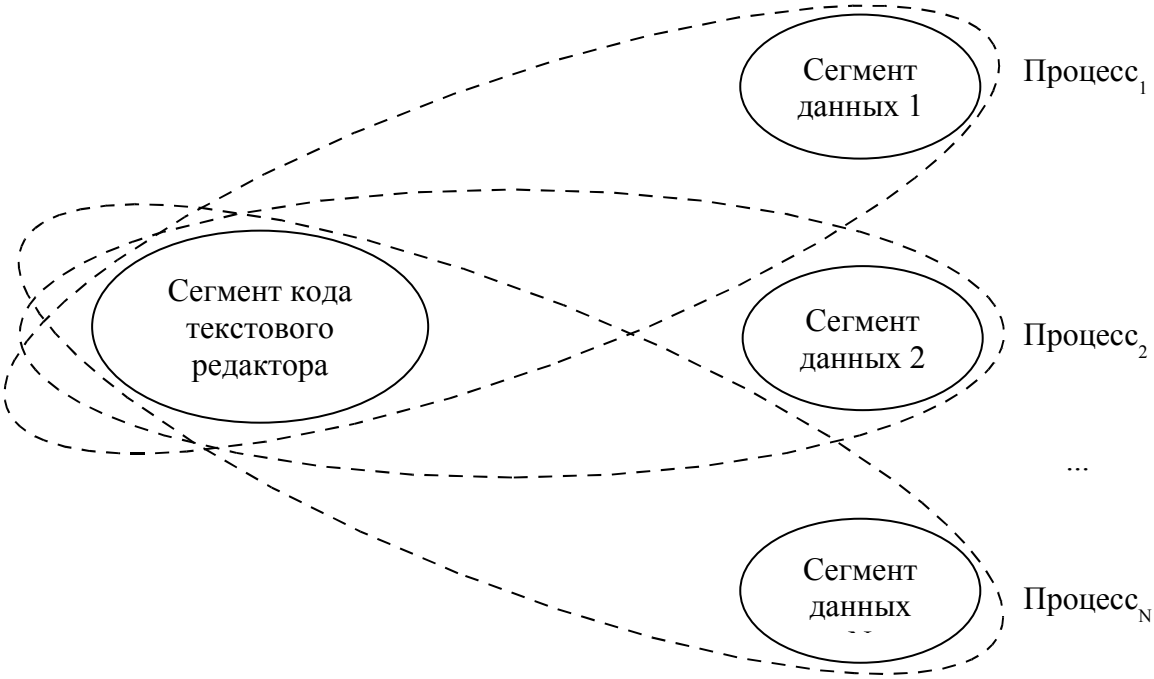
Тело процесса состоит из ***сегмента кода*** и ***сегмента данных***.

Сегмент кода содержит машинные команды и неизменяемые константы соответствующей процессу программы.

Сегмент данных – содержит данные, динамически изменяемые в ходе выполнения кода процесса. Сегмент данных содержит область статических переменных, область разделяемой с другими процессами памяти, а также область стека (обычно эта область служит основой для организации автоматических переменных, передачи параметров в функции, организацию динамической памяти).

Некоторые современные ОС имеют возможность разделения единого сегмента кода между разными процессами. Тем самым достигается экономия памяти в случаях одновременного выполнения идентичных процессов.

Например, при функционировании терминального класса одновременно могут быть сформированы несколько копий текстового редактора. В этом случае сегмент кода у всех процессов, соответствующих редакторам, будет единый, а сегменты данных будут у каждого процесса свои.
Следует отметить, что при использовании динамически загружаемых библиотек возможно разделение сегмента кода на неизменную часть, которая может разделяться между процессами и часть, соответствующую изменяемому в динамике коду подгружаемых



программ.

Аппаратная составляющая содержит все регистры и аппаратные таблицы ЦП, используемые активным или исполняемым процессом (счетчик команд, регистр состояния процессора, аппарат виртуальной памяти, регистры общего назначения и т. д.).

Обращаем внимание, что аппаратная составляющая имеет смысл только для процессов, находящихся в состоянии ***выполнения***. Для процессов, находящихся в других состояниях содержимое составляющей не определено.

Системная составляющая.
В системной составляющей контекста процесса содержатся различные атрибуты процесса, такие как:

- идентификатор родительского процесса;
- текущее состояние процесса;
- приоритет процесса;

- реальный идентификатор пользователя-владельца (идентификатор пользователя, сформировавшего процесс);
- эффективный идентификатор пользователя-владельца (идентификатор пользователя, по которому определяются права доступа процесса к файловой системе);
- реальный идентификатор группы, к которой принадлежит владелец (идентификатор группы к которой принадлежит пользователь, сформировавший процесс);
- эффективный идентификатор группы, к которой принадлежит владелец (идентификатор группы «эффективного» пользователя, по которому определяются права доступа процесса к файловой системе);
- список областей памяти;
- таблица открытых файлов процесса;
- информация о том, какая реакция установлена на тот или иной сигнал (аппарат сигналов позволяет передавать воздействия от ядра системы процессу и от процесса к процессу);
- информация о сигналах, ожидающих доставки в данный процесс;
- сохраненные значения аппаратной составляющей (когда выполнение процесса приостановлено).

Некоторые пояснения.
Реальные и эффективные идентификаторы пользователя и группы. Как правило при формировании процесса эти идентификаторы совпадают и равны реальному идентификатору пользователя и реальному идентификатору группы, т.е. они определяются персонификацией пользователя, сформировавшего данный процесс. При этом права процесса по доступу к файловой системе определяются правами сформировавшего процесс пользователя и его группы. Этого бывает недостаточно. Примером может служить ситуация, когда пользователь желает запустить некоторый процесс, изменяющий содержимое файлов, которые не принадлежать этому пользователю (например, изменение пароля на доступ пользователя в систему). Для разрешения данной ситуации имеется возможность установить специальный признак в исполняемом файле, наличие которого позволяет установить в процессе, сформированном при запуске данного файла в качестве эффективных идентификаторов, идентификатор владельца и группы владельца этого файла.

Рассмотрим второе определение процесса Unix.

Предварительно определим понятие ***системный вызов***. Системный вызов – специальная функция, позволяющая процессу обращаться к ядру ОС за выполнением тех или иных действий. Это может быть запрос на выполнение операций обмена, управления процессами, получения системной информации и т.п. При выполнении системного вызова в процессе происходит инициация специального прерывания ***«обращение к системе»***. В разных системах детали реализации системного вызова могут отличаться друг от друга и название прерывания ***«обращение к системе»*** здесь выбрано достаточно условно. В любом случае использование системного вызова влечет за собой накладные расходы, связанные со сменой контекста выполняющегося в данный момент процесса.

Если системный вызов не выполняется или выполняется нештатно, то он возвращает –1 в коде ответа и в переменной ***errno*** будет находится код причины отказа (для диагностирования результатов выполнения системного вызова в процессе используется переменная ***errno***, объявленная в файле ***errno.h***).

Процесс в ОС Unix – это объект, порожденный системным вызовом ***fork()***. Данный системный вызов является единственным стандартным средством порождения процессов в системе Unix. Ниже рассмотрим возможности данного системного вызова подробнее.

6.1.2 Базовые средства организации и управления процессами

Для порождения новых процессов в UNIX существует единая схема, с помощью которой создаются все процессы, существующие в работающем экземпляре ОС UNIX, за исключением первых двух процессов (0-го и 1-го)[1].
Для создания нового процесса в операционной системе UNIX используется системный вызов **fork(),** в результате в таблицу процессов заносится новая запись, и порожденный процесс получает свой уникальный идентификатор. Для нового процесса создается контекст, большая часть содержимого которого идентична контексту родительского процесса, в частности, тело порожденного процесса содержит копии сегментов кода и данных его родителя. Сыновний процесс наследует от родительского процесса:

- **окружение** - при формировании процесса ему передается некоторый набор параметров-переменных, используя которые, процесс может взаимодействовать с операционным окружением (интерпретатором команд и т.д.);
- **файлы, открытые в процессе-отце**, за исключением тех, которым было запрещено передаваться процессам-потомкам с помощью задания специального параметра при открытии. (Речь идет о том, что в системе при открытии файла с файлом ассоциируется некоторый атрибут, который определяет правила передачи этого открытого файла сыновним процессам. По умолчанию открытые в «отце» файлы можно передавать «потомкам», но можно изменить значение этого параметра и блокировать передачу открытых в процессе-отце файлов.);
- **способы обработки сигналов**;
- **разрешение переустановки эффективного идентификатора пользователя**;
- **разделяемые ресурсы** процесса-отца;
- **текущий рабочий каталог и домашний каталоги**
- и т.д.

По завершении системного вызова **fork()** каждый из процессов – родительский и порожденный – получив управление, продолжат выполнение с одной и той же инструкции одной и той же программы, а именно с той точки, где происходит возврат из системного вызова **fork().** Вызов **fork()** в случае удачного завершения возвращает сыновнему процессу значение **0**, а родительскому **PID** порожденного процесса. Это принципиально важно для различения сыновнего и родительского процессов, так как сегменты кода у них идентичны. Таким образом, у программиста имеется возможность разделить путь выполнения инструкций в этих процессах.

В случае неудачного завершения, т.е. если сыновний процесс не был порожден, системный вызов **fork()** возвращает –1, код ошибки устанавливается в переменной **errno**.

[1] Эти процессы создается во время начальной загрузки системы, механизм которой будет рассмотрен ниже

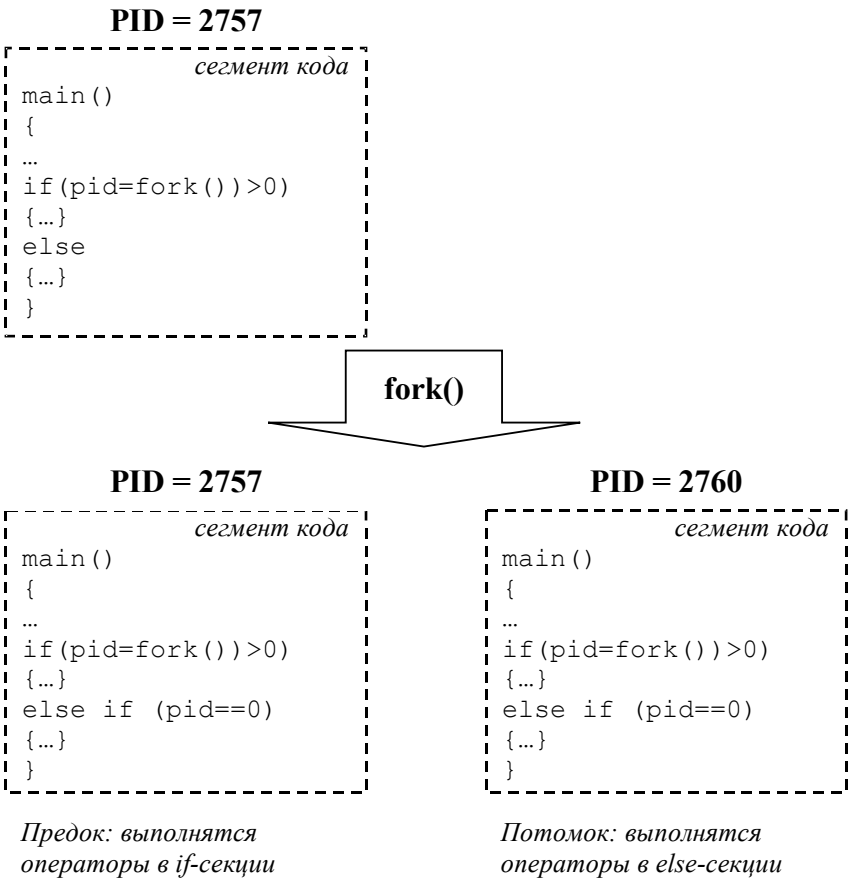


Рис. 2 Выполнение системного вызова fork()

Пример.

```
int main(int argc, char **argv)
{
      printf("PID=%d; PPID=%d \n",getpid(),getppid());
      /*печать PID  текущего процесса и PID процесса-предка */
      fork();
      /*создание нового процесса, с этого момента два процесса
      функционируют параллельно и независимо*/
      printf("PID=%d; PPID=%d \n",getpid(),getppid());
      /*оба процесса печатают PID  текущего процесса и PID процесса-
       предка*/
}
```

Пример.

Программа создает два процесса – процесс-предок распечатывает заглавные буквы, а процесс-потомок строчные.

```
int main(int argc, char **argv)
{
      char ch, first, last;
      int pid;
      if((pid=fork())>0)
      {
            /*процесс-предок*/
            first =’A’;
            last =’Z’;
      }
```

```
        else
        {
              /*процесс-потомок*/
              first ='a';
              last ='z';
        }

        for (ch = first; ch <= last; ch++)
        {
                    write(1,&ch,1);
        }
              _exit(0);
}
```

Механизм замены тела процесса.

Семейство системных вызовов **exec()** производит замену тела вызывающего процесса, после чего данный процесс начинает выполнять другую программу, передавая управление на точку ее входа. Возврат к первоначальной программе происходит только в случае ошибки при обращении к **exec()** , т.е. если фактической замены тела процесса не произошло.

Заметим, что выполнение "нового" тела происходит в рамках уже существующего процесса, т.е. после вызова **exec()** сохраняется идентификатор процесса, и идентификатор родительского процесса, таблица дескрипторов файлов, приоритет, и большая часть других атрибутов процесса. Фактически происходит замена сегмента кода и сегмента данных. Изменяются следующие атрибуты процесса:

- режимы обработки сигналов: для сигналов, которые перехватывались, после замены тела процесса будет установлена обработка по умолчанию, т.к. в новой программе могут отсутствовать указанные функции-обработчики сигналов;
- эффективные идентификаторы владельца и группы могут измениться, если для новой выполняемой программы установлен s-бит
- перед началом выполнения новой программы могут быть закрыты некоторые файлы, ранее открытые в процессе. Это касается тех файлов, для которых при помощи системного вызова **fcntl()** был установлен флаг **close-on-exec**. Соответствующие файловые дескрипторы будут помечены как свободные.

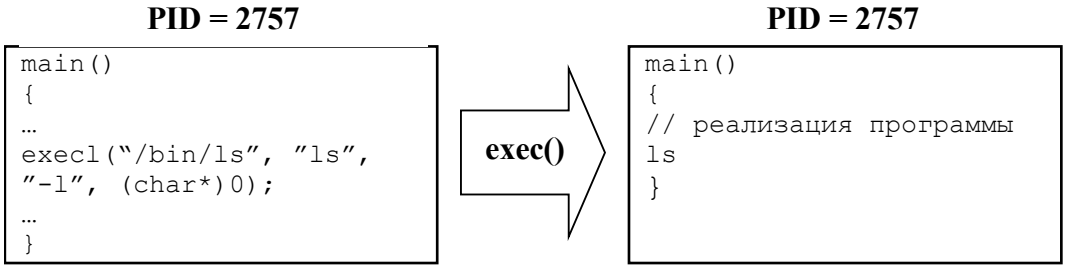


Рис. 3 Выполнение системного вызова exec()

Ниже представлены прототипы функций семейства **exec()**:

```
#include <unistd.h>
int execl(const char *path, char *arg0,…);
int execlp(const char *file, char *arg0,…);
int execle(const char *path, char *arg0,…, const char **env);
int execv(const char *path, const char **arg);
int execvp(const char *file, const char **arg);
int execve(const char *path, const char **arg, const char **env);
```

Первый параметр во всех вызовах задает имя файла программы, подлежащей исполнению. Этот файл должен быть исполняемым файлом и пользователь-владелец процесса должен иметь право на исполнение данного файла. Для функций с суффиксом «p» в названии имя файла может быть кратким, при этом при поиске нужного файла будет использоваться переменная окружения PATH. Далее передаются аргументы командной строки для вновь запускаемой программы, которые отобразятся в ее массив **argv** – в виде списка аргументов переменной длины для функций с суффиксом «l» либо в виде вектора строк для функций с суффиксом «v». В любом случае, в списке аргументов должно присутствовать как минимум 2 аргумента: имя программы, которое отобразится в элемент **argv[0]**, и значение NULL, завершающее список.

В функциях с суффиксом «e» имеется также дополнительный аргумент, описывающий переменные окружения для вновь запускаемой программы – это массив строк вида `name=value`, завершенный значением NULL.

Пример.

```
#include <unistd.h>
int main(int argc, char **argv)
{
      …
      /*тело программы*/
      …
      execl("/bin/ls","ls","-l",(char*)0);
      /*  или execlp("ls","ls", "-l",(char*)0);*/
      printf("это напечатается в случае неудачного обращения к предыдущей
      функции, к примеру, если не был найден файл ls \n");
      …
}
```

В данном случае второй параметр – вектор из указателей на параметры строки, которые будут переданы в вызываемую программу. Как и ранее первый указатель – имя программы, последний – нулевой указатель. Эти вызовы удобны, когда заранее неизвестно число аргументов вызываемой программы.

Пример.
Вызов программы компиляции.

```
int main(int argc, char **argv)
{
      char *pv[]={
                  "cc",
                  "-o",
                  "ter",
                  "ter.c",
                   (char*)0
                   };
      …
      /*тело программы*/
      …
      execv ("/bin/cc",pv);
      …
}
```

Чрезвычайно полезным является использование **fork()** совместно с системным вызовом **exec().** Как отмечалось выше системный вызов **exec()** используется для запуска исполняемого файла в рамках существующего процесса. Ниже приведена общая схема использования связки **fork() - exec().**

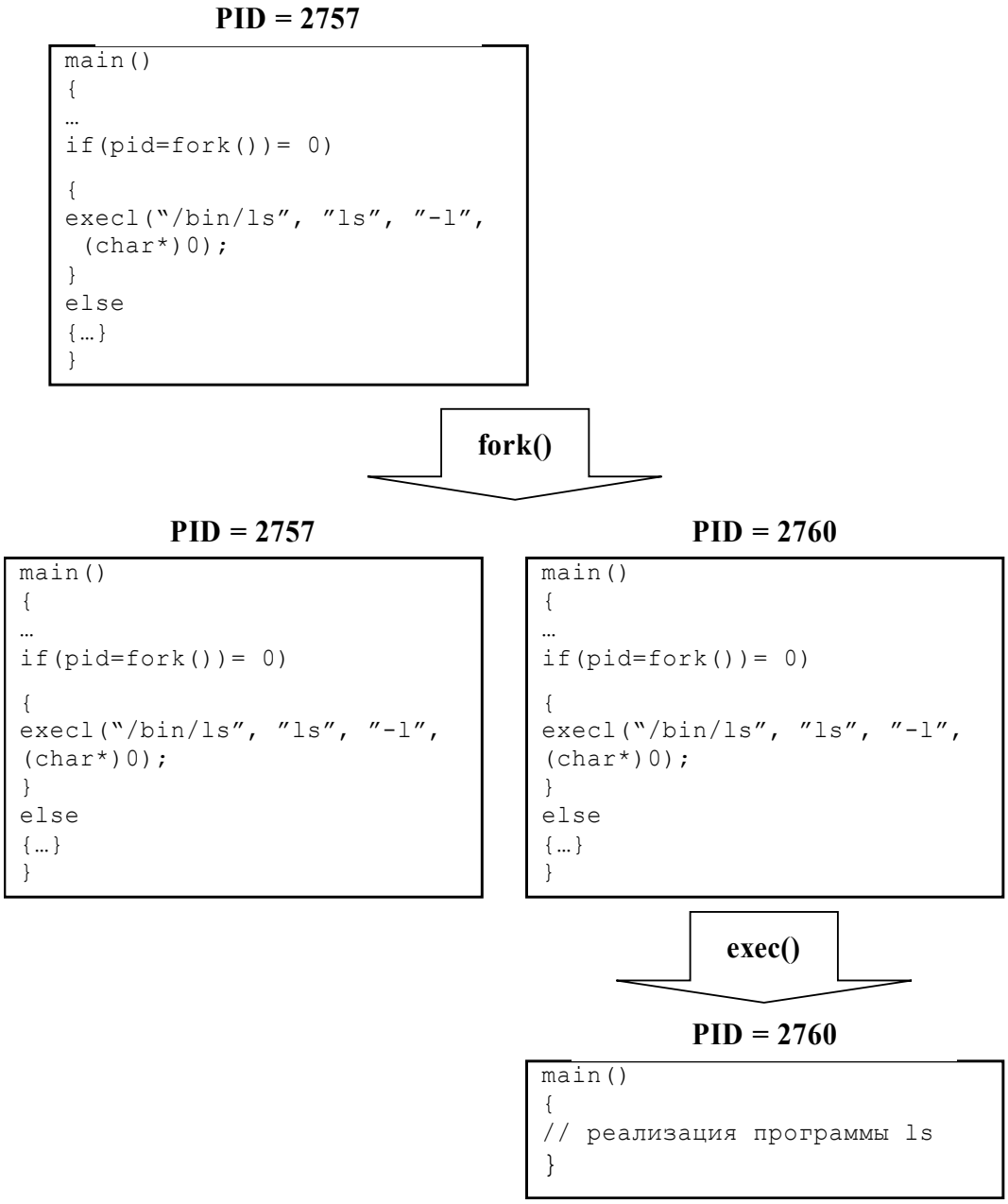


Рис. 4 Использование схемы fork()-exec()

Пример. Использование схемы fork-exec

Программа порождает три процесса, каждый из которых запускает программу **echo** посредством системного вызова **exec().** Данный пример демонстрирует важность проверки успешного завершения системного вызова **exec() ,** в противном случае возможно исполнение нескольких копий исходной программы. В нашем случае если все вызовы **exec()** проработают неуспешно, то копий программ будет восемь. Если все вызовы **exec()** будут успешными, то после последнего вызова **fork()** будет существовать четыре копии процесса. В каком порядке они пойдут на выполнение предсказать трудно.

```
int main(int argc, char **argv)
{
      if(fork()==0)
      {
            execl("/bin/echo","echo","это","сообщение один",NULL);
            printf("ошибка");
      }
      if(fork()==0)
      {
            execl("/bin/echo","echo","это","сообщение два",NULL);
            printf("ошибка");
      }
      if(fork()==0)
      {
            execl("/bin/echo","echo","это","сообщение три",NULL);
            printf("ошибка");
      }
      printf("процесс-предок закончился")
}
```

Результат работы может быть следующим:

процесс-предок закончился
это сообщение три
это сообщение два
это сообщение один

Завершение процесса.

Для завершения выполнения процесса предназначен системный вызов **_exit()**

```
void _exit(int exitcode);
```

Этот вызов никогда не завершается неудачно, поэтому для него не предусмотрено возвращающего значения. С помощью параметра **exit_code** процесс может передать породившему его процессу информацию о статусе своего завершения. Принято, хотя и не является обязательным правилом, чтобы процесс возвращал нулевое значение при нормальном завершении, и ненулевое – в случае какой-либо ошибки или нештатной ситуации.

Кроме обращения к вызову **_exit()**, другими причинами завершения процесса могут быть:

- оператора **return**, входящего в состав функции **main()**
- получение некоторых сигналов (об этом речь пойдет чуть ниже)

В любом из этих случаев происходит следующее:

- освобождаются сегмент кода и сегмент данных процесса
- закрываются все открытые дескрипторы файлов
- если у процесса имеются потомки, их предком назначается процесс с идентификатором 1
-
- освобождается большая часть контекста процесса, однако сохраняется запись в таблице процессов и та часть контекста, в которой хранится статус завершения процесса и статистика его выполнения
- процессу-предку завершаемого процесса посылается сигнал **SIGCHLD**

Состояние, в которое при этом переходит завершаемый процесс, в литературе часто называют состоянием “зомби”.

Процесс-предок имеет возможность получить информацию о завершении своего потомка. Для этого служит системный вызов **wait()**:

```
pid_t wait(int *status);
```

При обращении к этому вызову выполнение родительского процесса приостанавливается до тех пор, пока один из его потомков не завершится либо не будет остановлен. Если у процесса имеется несколько потомков, процесс будет ожидать завершения любого из них (т.е., если процесс хочет получить информацию о завершении каждого из своих потомков, он должен несколько раз обратиться к вызову **wait()**).

Возвращаемым значением **wait()** будет идентификатор завершенного процесса, а через параметр **status** будет возвращена информация о причине завершения процесса (путем вызова **_exit()** либо прерван сигналом) и коде возврата. Если процесс не интересуется это информацией, он может передать в качестве аргумента вызову **wait()** NULL-указатель.

Если к моменту вызова **wait()** один из потомков данного процесса уже завершился, перейдя в состояние зомби, то выполнение родительского процесса не блокируется, и **wait()** сразу же возвращает информацию об этом завершенном процессе. Если же к моменту вызова **wait()** у процесса нет потомков, системный вызов сразу же вернет –1. Также возможен аналогичный возврат из этого вызова, если его выполнение будет прервано поступившим сигналом.

После того, как информация о статусе завершения процесса-зомби будет доставлена его предку посредством вызова **wait()**, все оставшиеся структуры, связанные с данным процессом-зомби, освобождаются, и запись о нем удаляется из таблицы процессов. Таким образом, переход в состояние зомби необходим именно для того, чтобы процесс-предок мог получить информацию о судьбе своего завершившегося потомка, независимо от того, вызвал он **wait()** до или после его завершения.

Что происходит с процессом-потомком, если его предок вообще не обращался к **wait()** и/или завершился раньше потомка? Как уже говорилось, при завершении процесса отцом для всех его потомков становится процесс с идентификатором 1. Он и осуществляет системный вызов **wait(),** тем самым освобождая все структуры, связанные с потомками-зомби.

Часто используется сочетание функций **fork()-wait(),** если процесс-сын предназначен для выполнения некоторой программы, вызываемой посредством функции **exec().** Фактически этим предоставляется процессу- родителю возможность контролировать окончание выполнения процессов-потомков.

<u>Пример.</u> Использование системного вызова wait().

Пример программы, последовательно запускающей программы, имена которых указаны при вызове.

```
#include<stdio.h>
int main(int argc, char **argv)
{
      int i;
      for (i=1; i<argc; i++)
      {
            int status;
            if(fork()>0)
            {
                  /*процесс-предок ожидает сообщения
                    от процесса-потомка о завершении */
            wait(&status);
            printf("process-father\n");
            continue;
            }
     execlp(argv[i], argv[i], 0);
     exit();
   }
}
```

Пусть существуют три исполняемых файла **print1, print2, print3,** каждый из которых только печатает текст **first, second, third** соответственно, а код вышеприведенного примера находится в исполняемом файле с именем **file.** Тогда результатом работы команды **file print1, print2,print3** будет

first
process-father
second
process-father
third
process-father

Пример. Использование системного вызова wait().

В данном примере процесс-предок порождает два процесса, каждый из которых запускает команду **echo**. Далее процесс-предок ждет завершения своих потомков, после чего продолжает выполнение.

```
int main(int argc, char **argv)
{
      if ((fork()) == 0) /*первый процесс-потомок*/
      {     execl("/bin/echo","echo","this is","string 1",0);
            exit();     }
      if ((fork()) == 0) /*второй процесс-потомок*/
      {     execl("/bin/echo","echo","this is","string 2",0);
            exit();     }
      /*процесс-предок*/
      printf("process-father is waiting for children\n");
      while(wait() != -1);
      printf("all children terminated\n");
      exit();
}
```

В данном случае **wait()** вызывается в цикле три раза –первые два ожидают завершения процессов-потомков, последний вызов вернет неуспех, ибо ждать более некого.

6.1.3 Жизненный цикл процессов

Подведем некоторые итоги. Итак процесс представляет собой исполняемую программу вместе с необходимым ей окружением. Окружение состоит из информации о процессе, которая содержится в различных системных структурах данных, информации о содержимом регистров, программ операционной системы, стеке процесса, информации об открытых файлах, обработке сигналов и так далее. Процесс представляет собой изменяющийся во времени динамический объект. Программа представляет собой часть процесса. Процесс может создавать процессы-потомки посредством функции **fork(),** может изменять свою программу через системный вызов **exec().** Процесс может приостановить свое исполнение, используя функцию **wait()**, а также завершить свое исполнение посредством функции **exit().**

С учетом вышеизложенного рассмотрим подробнее состояния, в которых может находится процесс.

1. Процесс только что создан посредством вызова **fork()**.
2. Процесс находится в очереди готовых на выполнение процессов.
3. Процесс выполняется в режиме задачи, т.е. когда реализуется алгоритм, заложенный в программу. Выход из этого состояния может произойти через системный вызов, прерывание или завершение процесса.

4. Процесс может выполняться в режиме ядра ОС, т.е. когда по требованию процесса через системный вызов выполняются определенные инструкции ядра ОС или произошло другое прерывание.
5. Процесс в ходе выполнения не имеет возможность получить требуемый ресурс и переходит в состояние блокирования.
6. Процесс осуществил вызов **exit()** или получил сигнал на завершение. Ядро освобождает ресурсы, связанные с процессом, кроме кода возврата и статистики выполнения**.** Далее процесс переходит в состоянии *зомби,* а затем уничтожается.

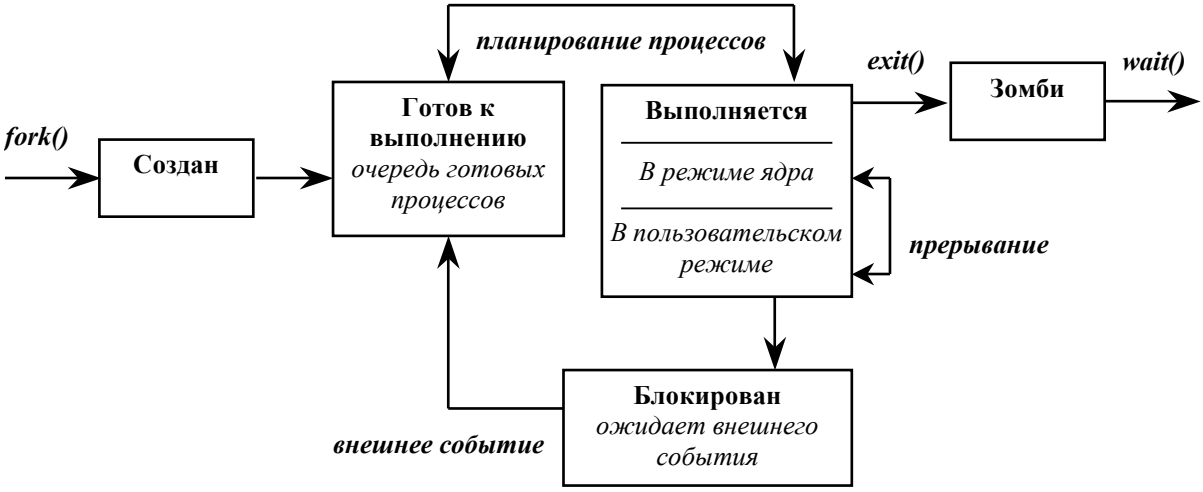


Рис. 5 Жизненный цикл процесса.

Дескрипторы открытых файлов
Сигналы
Специальные режимы и права процесса. S-бит, T-бит

6.1.4 Формирование процессов 0 и 1

Выше упоминалось о нестандартном формировании некоторых процессов в Unix. Речь шла о процессе начальной загрузки системы и нестандартном формировании двух специфических процессов с PID 0 и 1.

Рассмотрим подробнее, что происходит в момент начальной загрузки ОС UNIX. Начальная загрузка – это загрузка ядра системы в основную память и ее запуск. Нулевой блок каждой файловой системы предназначен для записи короткой программы, выполняющей начальную загрузку. Начальная загрузка выполняется в несколько этапов.
1.Аппаратный загрузчик читает нулевой блок системного устройства.
2.После чтения этой программы она выполняется, т.е. ищется и считывается в память файл **/unix**, расположенный в корневом каталоге и который содержит код ядра системы.
3.Запускается на исполнение этот файл.
В самом начале ядром выполняются определенные действия по инициализации системы, а именно, устанавливаются системные часы (для генерации прерываний), формируется диспетчер памяти, формируются значения некоторых структур данных (наборы буферов блоков, буфера индексных дескрипторов) и ряд других. По окончании этих действий происходит инициализация процесса с номером **"0".** По понятным причинам для этого невозможно использовать методы порождения процессов, изложенные выше, т.е. с использованием функций **fork()** и **exec().** При инициализации этого процесса резервируется память под его контекст и формируется нулевая запись в таблице процессов. Основными отличиями нулевого процесса являются следующие моменты

1.Данный процесс не имеет кодового сегмента , это просто структура данных, используемая ядром и процессом его называют потому, что он каталогизирован в таблице процессов.
2. Он существует в течении всего времени работы системы (чисто системный процесс) и считается, что он активен, когда работает ядро ОС.

Далее ядро копирует **"0"** процесс и создает **"1"** процесс. Алгоритм создания этого процесса напоминает стандартную процедуру, хотя и носит упрощенный характер. Сначала процесс **"1"** представляет собой полную копию процесса **"0"** , т.е. у него нет области кода. Далее происходит увеличение его размера и во вновь созданную кодовую область копируется программа, реализующая системный вызов **exec()** , необходимый для выполнения программы **/etc/init.** На этом завершается подготовка первых двух процессов. Первый из них представляет собой структуру данных, при помощи которой ядро организует мультипрограммный режим и управление процессами. Второй – это уже подобие реального процесса. Далее ОС переходит к выполнению программ диспетчера. Диспетчер наделен обычными функциями и на первом этапе у него нет выбора – он запускает **exec() ,** который заменит команды процесса **"1"** кодом, содержащимся в файле **/etc/init.** Получившийся процесс, называемый **init**, призван настраивать структуры процессов системы. Далее он подключает интерпретатор команд к системной консоли. Так возникает однопользовательский режим, так как консоль регистрируется с корневыми привилегиями и доступ по каким-либо другим линиям связи невозможен. При выходе из однопользовательского режима **init** создает многопользовательскую среду. С этой целью **init** организует процесс **getty** для каждого активного канала связи, т.е. каждого терминала Это программа ожидает входа кого-либо по каналу связи. Далее, используя системный вызов **exec(), getty** передает управление программе **login,** проверяющей пароль. Во время работы ОС процесс **init** ожидает завершения одного из порожденных им процессов, после чего он активизируется и создает новую программу **getty** для соответствующего терминала. Таким образом процесс **init** поддерживает многопользовательскую структуру во время функционирования системы. Схема описанного “круговорота” представлена на Рис. 6

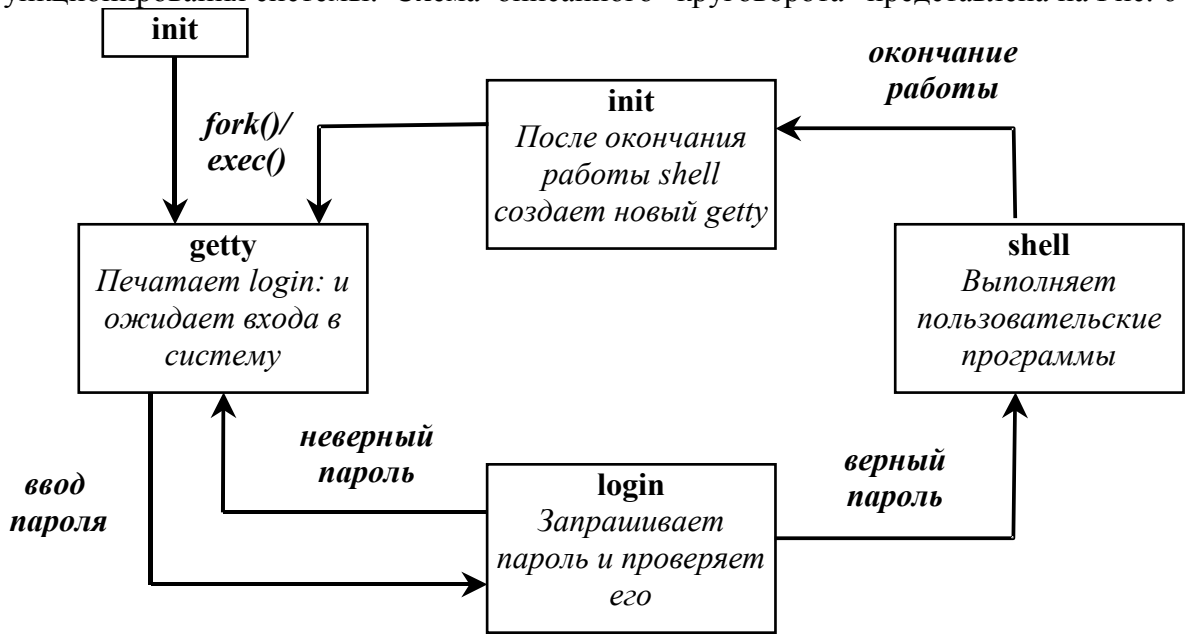


Рис. 6 Поддержка многопользовательской работы в ОС UNIX.

6.1.5 Планирование процессов. Свопинг

Планирование процесса, которому предстоит занять время центрального процессора, основывается на понятии приоритета. Каждому процессу сопоставляется некоторое целое числовое значение его приоритета (в т.ч., возможно, и отрицательное). Общее правило таково: чем больше числовое значение приоритета процесса, тем меньше его приоритет, т.е. наибольшие шансы занять время ЦП будут у того процесса, у которого числовое значение приоритета минимально.

Итак, числовое значение приоритета, или просто приоритет процесса - это параметр, который размещен в таблице процессов, и по значению этого параметра осуществляется выбор очередного процесса для продолжения работы и принимается решение о приостановке работающего процесса. Приоритеты системных и пользовательских процессов вычисляются по-разному. Рассмотрим, как это происходит для пользовательского процесса.

В вычислении приоритета **P_PRI** используются две изменяемые составляющие - **P_NICE** и **P_CPU. P_NICE** - это пользовательская составляющая приоритета. Его начальное значение полагается равным системной константе **NZERO**, в процессе выполнения процесса **P_NICE** может модифицироваться системным вызовом **nice().** Аргументом этого системного вызова является добавка к текущему значению (для обычного – непривилегированного - процесса эти добавки представляют собой неотрицательные числа). Значение **P_NICE** наследуется при порождении процессов, и таким образом, значение приоритета не может быть понижено при наследовании. Заметим, что изменяться **P_NICE** может только в сторону увеличения значения (до некоторого предельного значения), таким образом пользователь может снижать приоритет своих процессов.

P_CPU - это системная составляющая. Она формируется системой следующим образом: при прерывании по таймеру через предопределенные периоды времени для процесса, занимающего процессор в текущий момент, **P_CPU** увеличивается на единицу. Также, как и **P_NICE**, **P_CPU** имеет некоторое предельное значение. Если процесс будет находиться в состоянии выполнения так долго, что составляющая **P_CPU** достигнет своего верхнего предела, то значение **P_CPU** будет сброшено в нуль, а затем снова начнет расти. Отметим, однако, что такая ситуация весьма маловероятна, т.к. скорее всего, этот процесс будет выгружен и заменен другим еще до того момента, как **P_CPU** достигнет максимума.

В общем случае, приоритет процесса есть функция:

P_PRI = f(P_NICE, P_CPU)

Константа **P_USER** представляет собой нижний порог приоритета для пользовательских процессов. Пользовательская составляющая, как правило, учитывается в виде разности **_NICE – NZERO**, что позволяет принимать в расчет только добавку, введенную посредством системного вызова **nice().** Системная составляющая учитывается с некоторым коэффициентом. Поскольку неизвестно, проработал ли до момента прерывания процесс на процессоре полный интервал между прерываниями, то берется некоторое усреднение. Суммарно получается следующая формула для вычисления приоритета

P_PRI = P_USER + P_NICE – NZERO + P_CPU/a.

Заметим, что, если приоритет процесса не изменялся при помощи **nice()**, то единственной изменяемой составляющей приоритета будет **P_CPU**, причем эта составляющая растет только для того процесса, который находится в состоянии выполнения. В тот момент, когда значение ее станет таково, что в очереди готовых к выполнению процессов найдется процесс с меньшим значением приоритета, выполняемый процесс будет приостановлен и заменен

процессом с меньшим значением приоритета. При этом значение составляющей **P_CPU** для выгруженного процесса сбрасывается в нуль.

Пример. Рассмотрим два активных процесса, разделяющих процессор, причем таких, что ни их процессы-предки, ни они сами не меняли составляющую **P_NICE** системным вызовом **nice() .** Тогда **P_NICE =NZERO и** оба процесса имеют начальное значение приоритета **P_PRI = P_USER**, так как **P_CPU=0.** Пусть значение **P_CPU** увеличивается на единицу через **N** единиц времени (частота прерывания по таймеру), а в вычисление приоритета она входит с коэффициентом **1/A.** Таким образом, дополнительная единица в приоритете процесса, занимающего процессор, «набежит» через **A** таймерных интервалов. Значение **P_CPU** второго процесса остается неизменным, и его приоритет остается постоянным. Через **NA** единиц времени разница приоритетов составит единицу в пользу второго процесса и произойдет смена процессов на процессоре.

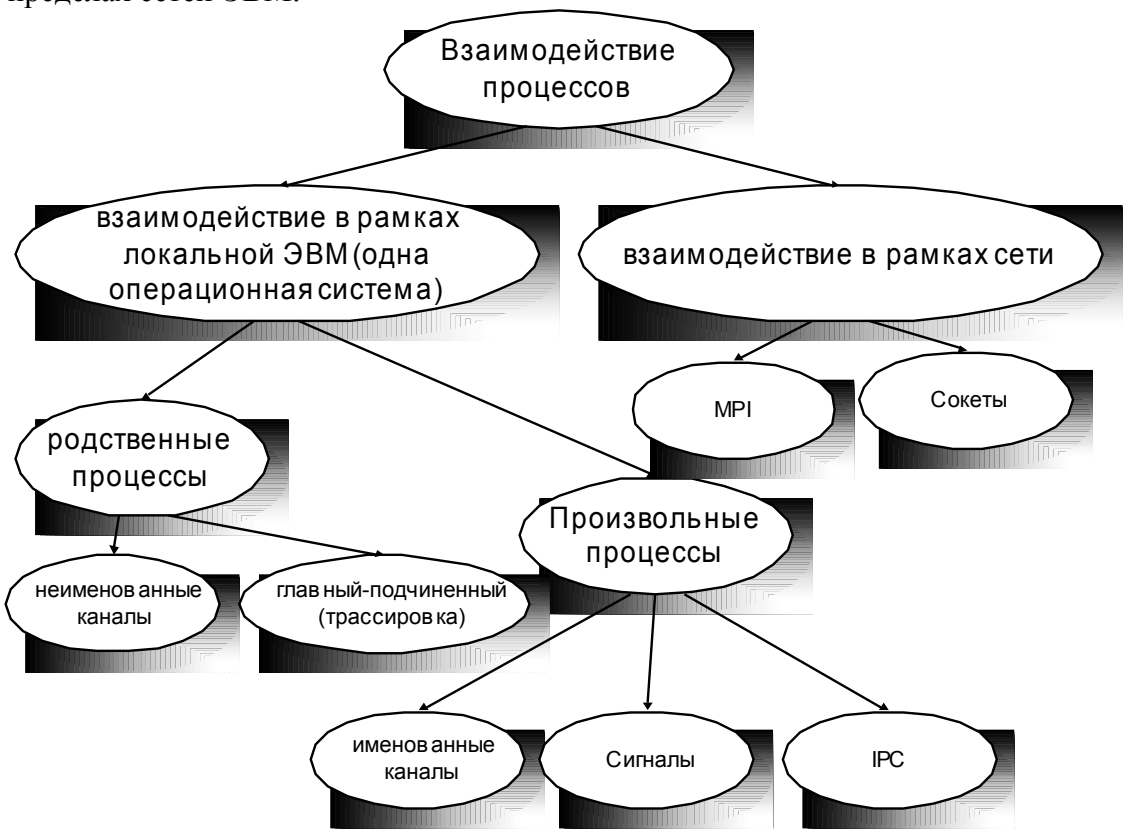
Принципы организация своппинга.

В системе определенным образом выделяется пространство для области свопинга. Есть пространство оперативной памяти, в котором находятся процессы, обрабатываемые системой в режиме мультипрограммирования. Есть область на ВЗУ, предназначенная для откачки этих процессов по мере необходимости. Упрощенная схема планирования подкачки основывается на использовании некоторого приоритета, который называется **P_TIME** и также находится в контексте процесса. В этом параметре аккумулируется время пребывания процесса в состоянии мультипрограммной обработки, или в области свопинга. В поле **P_TIME** существует счётчик выгрузки (**outage**) и счётчик загрузки (**inage**).

При перемещении процесса из оперативной памяти в область свопинга или обратно система обнуляет значение параметра **P_TIME**. Для загрузки процесса в память из области свопинга выбирается процесс с максимальным значением **P_TIME**. Если для загрузки этого процесса нет свободного пространства оперативной памяти, то система ищет среди процессов в оперативной памяти процесс, ожидающий ввода/вывода (сравнительно медленных операций, процессы у которых приоритет выше значения **P_ZERO**) и имеющий максимальное значение **P_TIME** (т.е. тот, который находился в оперативной памяти дольше всех). Если такого процесса нет, то выбирается просто процесс с максимальным значением **P_TIME**.

6.1.6 Механизмы взаимодействия процессов в ОС Unix. Основные концепции

Средства межпроцессного взаимодействия ОС Unix позволяют строить прикладные системы различной топологии, функционирующие, как в пределах одной локальной ЭВМ, так и в пределах сетей ЭВМ.



Будем акцентировать наше внимание на решении двух проблем, связанных с организацией взаимодействия процессов: именование взаимодействующих процессов и синхронизация процессов при организации взаимодействия.

Первая – именование процессов отправителей и получателей или именование некоторого объекта, через который осуществляется взаимодействие. Эта проблема решается по-разному в зависимости от конкретного механизма взаимодействия.

Так в системах, обеспечивающих взаимодействие процессов, функционирующих на различных компьютерах в сети используется адресация, принятая в конкретной сети ЭВМ (например, аппарат сокетов, MPI).

В средствах взаимодействия процессов, локализованных в пределах одной ЭВМ способ именования зависит от конкретного механизма взаимодействия. В частности, для ОС Unix взаимодействие процессов можно разделить на механизмы взаимодействия доступные исключительно родственным процессам и взаимодействие произвольных процессов (с точностью до прав процесса).

При взаимодействии родственных процессов проблема именования решается за счет наследования потомками некоторых свойств одного из прародителей. Например, в случае неименованных каналов процесс-родитель для организации взаимодействия создает канал.

Этот канал наследуется сыновними процессами, тем самым создается возможность организации симметричного (ибо все процессы изначально равноправны) взаимодействия родственных процессов. Другой пример, это взаимодействие процессов по схеме главный-подчиненный (или трассировка). Данный тип взаимодействия ассиметричный, так как изначально один из взаимодействующих процессов получает статус и права «главного», второй - «подчиненного». Главный – это родительский процесс, подчиненный – сыновний. Соответственно именование жестко привязано к связке отец-сын (идентификаторы сына и отца всегда доступны и однозначно определены).

При взаимодействии произвольных процессов нет факта наследования некоторых свойств процессов, которые могут использоваться для именования. Поэтому, в данном случае обычно используются две схемы. Первая – использование для именования идентификаторов взаимодействующих процессов (к примеру, аппарат передачи сигналов). Вторая схема предполагает использование некоторого системного ресурса, обладающего уникальным именем. Примером могут являться именованные каналы, использующие для организации взаимодействия процессов файлы специальных типов (например, FIFO).

Другая проблема организации взаимодействия – это проблема синхронизации взаимодействующих процессов. Суть проблемы состоит в следующем. Взаимодействие процессов представимо в виде оказания одним процессом воздействия на другой процесс или использование некоторых разделяемых ресурсов, через которые возможна организация обмена данными.

Первое требование к средствам взаимодействия процессов это атомарность (неразделяемость) базовых операций. То есть синхронизация должна обеспечить атомарность операций взаимодействий или обмена данными с разделяемыми ресурсами. К примеру, система должна блокировать начало чтения данных из некоторого разделяемого ресурса до того, пока начавшаяся к этому моменту операция записи по этому ресурсу не завершится.

Второе требование – это обеспечение определенного порядка в операциях взаимодействия. Назовем это семантической синхронизацией. Например, попытка чтения данных, которых еще нет (и операция записи которых еще не начиналась). Уровней семантической синхронизации может быть достаточно много.

Комплексное решение проблемы синхронизации зависит от свойств используемых средств взаимодействия процессов. В некоторых случаях операционная система обеспечивает некоторые уровни синхронизации (например передача сигналов, использование каналов). В некоторых участие операционной системы в синхронизации минимально (например, разделяемая память IPC).

Но в любом случае, конкретная прикладная система должна учитывать, и при необходимости обеспечивать семантическую синхронизацию процессов.

6.2 Взаимодействие процессов в Unix, Базовые средства.

6.2.1 Сигналы.

Сигналы представляют собой средство уведомления процесса о наступлении некоторого события в системе. Инициатором посылки сигнала может выступать как другой процесс, так и сама ОС. Сигналы, посылаемые ОС, уведомляют о наступлении некоторых строго предопределенных ситуаций (как, например, завершение порожденного процесса, прерывание процесса нажатием комбинации Ctrl-C, попытка выполнить недопустимую машинную инструкцию, попытка недопустимой записи в канал и т.п.), при этом каждой такой ситуации сопоставлен свой сигнал. Кроме того, зарезервировано один или несколько номеров сигналов, семантика которых определяется пользовательскими процессами по своему усмотрению (например, процессы могут посылать друг другу сигналы с целью синхронизации).

Количество различных сигналов в современных версиях UNIX около 30, каждый из них имеет уникальное имя и номер. Описания представлены в файле **<signal.h>.** Ниже приведено несколько примеров[2].

2 - SIGINT /*прерывание*/
3 - SIGQUIT /*аварийный выход*/
9 - SIGKILL /*уничтожение процесса*/
14 - SIGALRM /*прерывание от таймера*/.
18 - SIGCHLD /*процесс-потомок завершился*/.

Сигналы являются механизмом асинхронного взаимодействия, т.е. момент прихода сигнала процессу заранее неизвестен. Однако процесс может предвидеть возможность получения того или иного сигнала и установить определенную реакцию на его приход. В этом плане сигналы можно рассматривать как программный аналог аппаратных прерываний.

При получении сигнала процессом возможны три варианта реакции на полученный сигнал:

- Процесс реагирует на сигнал стандартным образом, установленным по умолчанию (для большинства сигналов действие по умолчанию – это завершение процесса).
- Процесс может установить специальную обработку сигнала, в этом случае по приходу сигнала вызывается функция-обработчик, определенная процессом (при этом говорят, что сигнал перехватывается)
- Процесс может проигнорировать сигнал.

Для каждого сигнала процесс может устанавливать свой вариант реакции, например, некоторые сигналы он может игнорировать, некоторые перехватывать, а на остальные установить реакцию по умолчанию. При этом в процессе свое работы процесс может изменять вариант реакции на тот или иной сигнал. Однако, необходимо отметить, что некоторые сигналы невозможно ни перехватить, ни игнорировать. Они используются ядром ОС для управления работой процессов (например, **SIGKILL, SIGSTOP**).

Если в процесс одновременно доставляется несколько различных сигналов, то порядок их обработки не определен. Если же обработки ждут несколько экземпляров одного и того же сигнала, то ответ на вопрос, сколько экземпляров будет доставлено в процесс – все или один – зависит от конкретной реализации ОС.

Отдельного рассмотрения заслуживает ситуация, когда сигнал приходит в момент выполнения системного вызова. Обработка такой ситуации в разных версиях UNIX реализована по-разному, например, обработка сигнала может быть отложена до завершения системного вызова; либо системный вызов автоматически перезапускается после его прерывания сигналом; либо системный вызов вернет –1, а в переменной **errno** будет установлено значение **EINTR**

Для отправки сигнала существует системный вызов **kill()**:

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

[2] Следует заметить, что в разных версиях UNIX имена сигналов могут различаться.

```
#INCLUDE <SIGNAL.H>
INT KILL (PIT_T PID, INT SIG)
```

Первым параметром вызова служит идентификатор процесса, которому посылается сигнал (в частности, процесс может послать сигнал самому себе). Существует также возможность одновременно послать сигнал нескольким процессам, например, если значение этого параметра есть 0, сигнал будет передан всем процессам, которые принадлежат той же группе, что и процесс, посылающий сигнал, за исключением процессов с идентификаторами 0 и 1.

Во втором параметре передается номер посылаемого сигнала. Если этот параметр равен 0, то будет выполнена проверка корректности обращения к **kill()** (в частности, существование процесса с идентификатором **pid**), но никакой сигнал в действительности посылаться не будет.

Если процесс-отправитель не обладает правами привилегированного пользователя, то он может отправить сигнал только тем процессам, у которых реальный или эффективный идентификатор владельца процесса совпадает с реальным или эффективным идентификатором владельца процесса-отправителя.

Для определения реакции на получение того или иного сигнала в процессе служит системный вызов **signal()**:

```
#INCLUDE <SIGNAL.H>
void (*signal  ( int sig, void (*disp) (int))) (int)
```

где аргумент **sig** — номер сигнала, для которого устанавливается реакция, а **disp** — либо определенная пользователем функция-обработчик сигнала, либо одна из констант: **SIG_DFL** и **SIG_IGN.** Первая из них указывает, что необходимо установить для данного сигнала обработку по умолчанию, т.е. стандартную реакцию системы, а вторая — что данный сигнал необходимо игнорировать. При успешном завершении функция возвращает указатель на предыдущий обработчик данного сигнала (он может использоваться процессом, например, для восстановления прежней реакции на сигнал).

Как видно из прототипа вызова **signal()**, определенная пользователем функция-обработчик сигнала должна принимать один целочисленный аргумент (в нем будет передан номер обрабатываемого сигнала), и не возвращать никаких значений.

Отметим одну особенность реализации сигналов в ранних версиях UNIX: каждый раз при получении сигнала его диспозиция (т.е. действие при получении сигнала) сбрасывается на действие по умолчанию, т.о. если процесс желает многократно обрабатывать сигнал своим собственным обработчиком, он должен каждый раз при обработке сигнала заново устанавливать реакцию на него (см. пример 9)

В заключении отметим, что сигналы достаточно ресурсоемки, ибо отправка сигнал представляет собой системный вызов, а доставка сигнала - прерывание процесса-получателя. Вызов функции-обработчика и возврат требует операций со стеком. Сигналы также несут весьма ограниченную информацию.

Пример. Обработка сигнала.

В данном примере при получении сигнала **SIGINT** четырежды вызывается специальный обработчик, а в пятый раз происходит обработка по умолчанию.

```
      #include <sys/types.h>
      #include <signal.h>
      #include <stdio.h>
      int count=1;
void SigHndlr (int s)   /* обработчик сигнала */
      {
            printf("\n I got SIGINT %d time(s) \n", count ++);
            if (count==5) signal (SIGINT, SIG_DFL);
            /* ставим обработчик сигнала по умолчанию */
            else signal (SIGINT, SigHndlr);
```

```
            /* восстанавливаем обработчик сигнала */
      }

int main(int argc, char **argv)
      {
            signal (SIGINT, SigHndlr); /* установка реакции на сигнал */
            while (1); /*”тело программы” */
            return 0;
      }
```

Пример. Удаление временных файлов при завершении программы.

При разработке программ нередко приходится создавать временные файлы , которые позже удаляются. Если произошло непредвиденное событие, такие файлы могут остаться неудаленными. Ниже приведено решение этой задачи.

```
      #include <unistd.h>
      #include <signal.h>
      #include <sys/types.h>
      #include <sys/stat.h>
      #include <fcntl.h>
      const char * tempfile = “abc”;

void SigHndlr (int s)
      {
            unlink(tempfile);
            /* уничтожение временного файла в случае прихода сигнала
      SIGINT. В случае, если такой файл не существует (еще не создан или
      уже удален), вызов вернет -1 */
      }

int main(int argc, char **argv)
      {
            signal(SIGINT, SigHndlr); /*установка реакции на сигнал */
            …
            creat(tempfile, 0666); /*создание временного файла*/
            …
            unlink(tempfile);
            /*уничтожение временного файла в случае нормального
            функционирования процесса */
            return 0;
      }
```

В данном примере для создания временного файла используется системный вызов **creat()**:

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

А системный вызов **unlink()** удаляет имя и файл, на который оно ссылается.

```
#include <unistd.h>
int unlink(const char *pathname);
```

Пример. Программа “Будильник”.

Программа “Будильник”. Средствами ОС “заводится” будильник, который будет поторапливать ввести некоторое имя. Системный вызов **alarm():**

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int seconds);
```
инициализирует появление сигнала **SIGALRM** - процесс запрашивает ядро отправить сигнал по прошествии определенного времени**.**

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>



void alrm (int s) /*обработчик сигнала SIG_ALRM */
{
      printf("\n жду имя \n");
      alarm(5); /* заводим будильник */
      signal (SIGALRM,alrm); /* перестанавливаем реакцию на сигал*/
}

int main(int argc, char **argv)
{
      char s[80];
      signal(SIGALRM, alrm);
      /* установка обработчика alrm на приход сигнала SIG_ALRM */
      alarm(5); /* заводим будильник */
      printf("Введите имя \n");
      for (;;)
      {
            printf("имя:");
                  if (gets(s) != NULL) break; /* ожидаем ввода имени */
      };
      printf("OK! \n");
      return 0;
}
```

В начале программы мы устанавливаем реакцию на сигнал **SIGALRM** - функцию **alarm(),** далее мы заводим будильник, запрашиваем "***Введите имя***" и ожидаем ввода строки символов. Если ввод строки задерживается, то будет вызвана функция **alarm()**, которая напомнит, что программа "ждет имя", опять заведет будильник и поставит себя на обработку сигнала **SIGALRM** еще раз. И так будет до тех пор, пока не будет введена строка. Здесь имеется один нюанс: если в момент выполнения системного вызова возникает событие, связанное с сигналом, то система прерывает выполнение системного вызова и возвращает код ответа, равный «-1».

Пример. Двухпроцессный вариант программы "Будильник".

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

void alr(int s)
{
      printf("\n Быстрее!!! \n");
      signal (SIGALRM, alr);
      /* переустановка обработчика alr на приход сигнала SIGALRM */
}
```

```
int main(int argc, char **argv)
{
      char s[80];
      int pid;

      signal(SIGALRM, alr);
      /* установка обработчика alr на приход сигнала SIGALRM */
      if  (pid=fork()) {
            for (;;)
            {
                  sleep(5);  /*приостанавливаем процесс на 5 секунд */
                  kill(pid, SIGALRM);
                  /*отправляем сигнал SIGALRM процессу-сыну */
            }
      }
      else {
            printf("Введите имя \n");
            for (;;)
            {
                  printf("имя:");
                        if  ( gets(s)  !=  NULL  )  break;  /*ожидаемм  ввода
                        имени*/
            }
            printf("OK!\n");
            kill(getppid(), SIGKILL);
            /* убиваем зациклившегося отца */
      }
      return 0;
}
```

В данном случае программа реализуется в двух процессах. Как и в предыдущем примере, имеется функция реакции на сигнал **alr(),** которая выводит на экран сообщение и переустанавливает функцию реакции на сигнал, опять же на себя. В основной программе мы также указываем **alr()** как реакцию на **SIGALRM**. После этого мы запускаем сыновний процесс, и отцовский процесс (бесконечный цикл) "засыпает" на 5 единиц времени, после чего сыновнему процессу будет отправлен сигнал **SIGALRM.** Все, что ниже цикла, будет выполняться в процессе-сыне: мы ожидаем ввода строки, если ввод осуществлен, то происходит уничтожение отца (**SIGKILL**).

6.2.2 Неименованные каналы.

Одним из простейших средств взаимодействия процессов в операционной системе UNIX является механизм каналов. Неименованный канал есть некая сущность, в которую можно помещать и извлекать данные, для чего служат два файловых дескриптора, ассоциированных с каналом: один для записи в канал, другой — для чтения. Для создания канала служит системный вызов **pipe():**

```
INT PIPE (INT *FD)
```

Данный системный вызов выделяет в оперативной памяти некоторое ограниченное пространство и возвращает чебрез параметр fd массив из двух файловых дескрипторов: один для записи в канал — **fd[1],** другой для чтения — **fd[0].**

Эти дескрипторы являются дескрипторами открытых файлов, с которыми можно работать, используя такие системные вызовы как **read()**, **write(), dup()** и пр**.**

Однако существуют различия в организации использования обычного файла и канала.

Особенности организации чтения данных из канала:

- если прочитано меньше байтов, чем находится в канале, оставшиеся сохраняются в канале;
- если делается попытка прочитать больше данных, чем имеется в канале, и при этом существуют открытые дескрипторы записи, ассоциированные с каналом, будет прочитано (т.е. изъято из канала) доступное количество данных, после чего читающий процесс блокируется до тех пор, пока в канале не появится достаточное количество данных для завершения операции чтения;
- процесс может избежать такого блокирования, изменив для канала режим блокировки с использованием системного вызова **fcntl()**, в этом случае будет считано доступное количество данных, и управление будет сразу возвращено процессу;
- при закрытии записывающей стороны канала, в него помещается символ EOF (т.е. ситуация когда закрыты все дескрипторы, ассоциированные с записью в канал), после этого процесс, осуществляющий чтение, может выбрать из канала все оставшиеся данные и признак конца файла, благодаря которому блокирования при чтении в этом случае не происходит.

Особенности организации записи данных в канал:

- если процесс пытается записать большее число байтов, чем помещается в канал (но не превышающее предельный размер канала) записывается возможное количество данных, после чего процесс, осуществляющий запись, блокируется до тех пор, пока в канале не появится достаточное количество места для завершения операции записи;
- процесс может избежать такого блокирования, изменив для канала режим блокировки с использованием системного вызова **fcntl()**. В неблокирующем режиме в ситуации, описанной выше, будет записано возможное количество данных, и управление будет сразу возвращено процессу.
- если же процесс пытается записать в канал порцию данных, превышающую предельный размер канала, то будет записано доступное количество данных, после чего процесс заблокируется до появления в канале свободного места любого размера (пусть даже и всего 1 байт), затем процесс разблокируется, вновь производит запись на доступное место в канале, и если данные для записи еще не исчерпаны, вновь блокируется до появления свободного места и т.д., пока не будут записаны все данные, после чего происходит возврат из вызова **write()**
- если процесс пытается осуществить запись в канал, с которым не ассоциирован ни один дескриптор чтения, то он получает сигнал **SIGPIPE** (тем самым ОС уведомляет его о недопустимости такой операции)**.**

В стандартной ситуации (при отсутствии переполнения) система гарантирует атомарность операции записи, т. е. при одновременной записи нескольких процессов в канал их данные не перемешиваются.

Пример. Использование канала.

Пример использования канала в рамках одного процесса – копирование строк. Фактически осуществляется посылка данных самому себе.

```
int main(int argc, char **argv)
{
      char s*=”chanel”;
      char buf[80];
      int pipes[2];
      pipe(pipes);
      write(pipes[1],s,strlen(s)+1);
      read(pipes[0],buf,strlen(s)+1);
      close(pipes[0]);
      close(pipes[1]);
      printf(“%s\n”,buf);
}
```

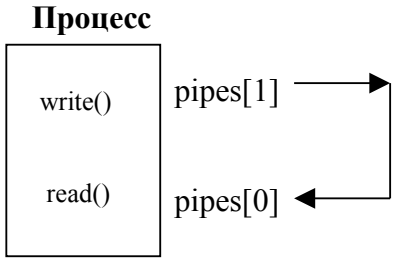


Рис. 9 Обмен через канал в рамках одного процесса.

Чаще всего, однако, канал используется для обмена данными между несколькими процессами. При организации такого обмена используется тот факт, что при порождении сыновнего процесса посредством системного вызова **fork()** наследуется таблица файловых дескрипторов процесса-отца, т.е. все файловые дескрипторы, доступные процессу-отцу, будут доступны и процессу-сыну. Таким образом, если перед порождением потомка был создан канал, файловые дескрипторы для доступа к каналу будут унаследованы и сыном. В итоге обоим процессам оказываются доступны дескрипторы, связанные с каналом, и они могут использовать канал для обмена данными (см. рис. 10 и пример 14).

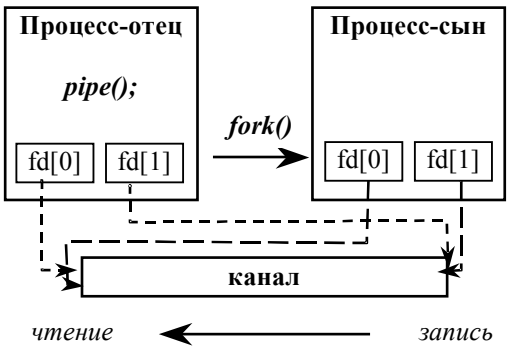


Рис. 10 Пример обмена данными между процессами через канал.

Пример. Схема взаимодействия процессов с использованием канала.

```
int main(int argc, char **argv)
{    int fd[2];
     pipe(fd);
     if(fork())
     {/*процесс-родитель*/
          close(fd[0]); /* закрываем ненужный дескриптор */
          write (fd[1], …);
          …
          close(fd[1]);
          …
     }
     else
     {/*процесс-потомок*/
          close(fd[1]); /* закрываем ненужный дескриптор */
          while(read (fd[0], …))
          {
               …
          }
          …
     }
}
```

Аналогичным образом может быть организован обмен через канал между двумя потомками одного порождающего процесса и вообще между любыми родственными процессами, единственным требованием здесь, как уже говорилось, является необходимость создавать канал в порождающем процессе прежде, чем его дескрипторы будут унаследованы порожденными процессами.

Как правило, канал используется как однонаправленное средство передачи данных, т.е. только один из двух взаимодействующих процессов осуществляет запись в него, а другой процесс осуществляет чтение[3], при этом каждый из процессов закрывает не используемый им дескриптор. Это особенно важно для неиспользуемого дескриптора записи в канал, т.к. именно при закрытии пишущей стороны канала в него помещается символ конца файла. Если, к примеру, в рассмотренном примере 14 процесс-потомок не закроет свой дескриптор записи в канал, то при последующем чтении из канала, исчерпав все данные из него, он будет заблокирован, т.к. записывающая сторона канала будет открыта, и следовательно, читающий процесс будет ожидать очередной порции данных.

[3] это правило не является обязательным, но для корректной организации двустороннего обмена через один канал требуется дополнительная синхронизация

Пример. Реализация конвейера.

Пример реализации конвейера **print|wc –** вывод программы **print** будет подаваться на вход программы **wc.** Программа **print** печатает некоторый текст. Программа **wc** считает количество прочитанных строк, слов и символов.

```
#include <stdio.h>
int main(int argc, char **argv)
{
      int fd[2];
      pipe(fd);  /*организован канал*/
      if(fork())
      {
            /*процесс-родитель*/
              dup2(fd[1],1);
            /*отождествили стандартный вывод с файловым дескриптором
            канала, предназначенным для записи*/
                close(fd[1]);      /*закрыли файловый дескриптор канала,
            предназначенный для записи */
                close(fd[0]);      /*закрыли файловый дескриптор канала,
            предназначенный для чтения */
              exelp("print","print",0); /*запустили программу print */
      }
      /*процесс-потомок*/
      dup2(fd[0],0);
            /*отождествили стандартный ввод с файловым дескриптором
            канала,   предназначенным для чтения*/
      close(fd[0]);
            /* закрыли файловый дескриптор канала, предназначенный для
            чтения */
      close(fd[1]);
            /* закрыли файловый дескриптор канала, предназначенный для
            записи */
      execl("/usr/bin/wc","wc",0); /* запустили программу wc */
}
```

Пример. Совместное использование сигналов и каналов – «пинг-понг».
Пример программы с использованием каналов и сигналов для осуществления связи между процессами – весьма типичной ситуации в системе. При этом на канал возлагается роль среды двусторонней передачи информации, а на сигналы – роль системы синхронизации при передаче информации. Процессы посылают друг другу целое число, всякий раз увеличивая его на 1. Когда число достигнет некоего максимума, оба процесса завершаются.

```
      #include <signal.h>
      #include <sys/types.h>
      #include <sys/wait.h>
      #include <unistd.h>
      #include <stdlib.h>
      #include <stdio.h>
      #define MAX_CNT 100
      int target_pid, cnt;
      int fd[2];
      int status;
void SigHndlr (int s)
      {/* в обработчике сигнала происходит и чтение, и запись */
      signal(SIGUSR1, SigHndlr);
      if (cnt < MAX_CNT)
      {     read(fd[0], &cnt, sizeof(int));
            printf("%d \n", cnt);
            cnt++;
            write(fd[1], &cnt, sizeof(int));
            /* посылаем сигнал второму: пора читать из канала */
            kill(target_pid, SIGUSR1);
      }
      else
            if (target_pid == getppid())
            {     /* условие окончания игры проверяется потомком */
                  printf("Child is going to be terminated\n");
                  close(fd[1]); close(fd[0]);
                  /* завершается потомок */
                  exit(0);
            } else kill(target_pid, SIGUSR1);
      }
int main(int argc, char **argv)
{     pipe(fd); /* организован канал */
      signal(SIGUSR1, SigHndlr);
            /* установлен обработчик сигнала для обоих процессов */
      cnt = 0;
      if (target_pid = fork())
      {     /* Предку остается только ждать завершения потомка */
            wait(&status);
            printf("Parent is going to be terminated\n");
            close(fd[1]); close(fd[0]);
            return 0;
      }
      else
      {     /* процесс-потомок узнает PID родителя */
            target_pid = getppid();
            /* потомок начинает пинг-понг */
            write(fd[1], &cnt, sizeof(int));
            kill(target_pid, SIGUSR1);
            for(;;); /* бесконечный цикл */
      }
}
```

6.2.3 Именованные каналы.

Рассмотренные выше программные каналы имеют важное ограничение: т.к. доступ к ним возможен только посредством дескрипторов, возвращаемых при порождении канала, необходимым условием взаимодействия процессов через канал является передача этих дескрипторов по наследству при порождении процесса. Именованные каналы (FIFO-файлы) расширяют свою область применения за счет того, что подключиться к ним может любой процесс в любое время, в том числе и после создания канала. Это возможно благодаря наличию у них имен**.**

FIFO-файл представляет собой отдельный тип файла в файловой системе UNIX, который обладает всеми атрибутами файла, такими как имя владельца, права доступа и размер. Для его создания в UNIX System V.3 и ранее используется системный вызов **mknod(),** а в BSD UNIX и System V.4 – вызов **mkfifo()** (этот вызов поддерживается и стандартом POSIX):

```
INT MKNOD (CHAR *PATHNAME, MODE_T MODE, DEV)
INT MKFIFO (CHAR *PATHNAME, MODE_T MODE)
```

В обоих вызовах первый аргумент представляет собой имя создаваемого канала, во втором указываются права доступа к нему для владельца, группы и прочих пользователей, и кроме того, устанавливается флаг, указывающий на то, что создаваемый объект является именно FIFO-файлом (в разных версиях ОС он может иметь разное символьное обозначение – **S_IFIFO** или **I_FIFO**). Третий аргумент вызова **mknod()** игнорируется.

После создания именованного канала любой процесс может установит с ним связь посредством системного вызова **open()**. При этом действуют следующие правила:

- если процесс открывает FIFO-файл для чтения, он блокируется до тех пор, пока какой-либо процесс не откроет тот же канал на запись
- если процесс открывает FIFO-файл на запись, он будет заблокирован до тех пор, пока какой-либо процесс не откроет тот же канал на чтение
- процесс может избежать такого блокирования, указав в вызове **open()** специальный флаг (в разных версиях ОС он может иметь разное символьное обозначение – **O_NONBLOCK** или **O_NDELAY**). В этом случае в ситуациях, описанных выше, вызов **open()** сразу же вернет управление процессу

Правила работы с именованными каналами, в частности, особенности операций чтения-записи, полностью аналогичны неименованным каналам.

Ниже рассматривается пример, где один из процессов является сервером, предоставляющим некоторую услугу, другой же процесс, который хочет воспользоваться этой услугой, является клиентом. Клиент посылает серверу запросы на предоставление услуги, а сервер отвечает на эти запросы.

Пример.

Процесс-сервер запускается на выполнение первым, создает именованный канал, открывает его на чтение в неблокирующем режиме и входит в цикл, пытаясь прочесть что-либо. Затем запускается процесс-клиент, подключается к каналу с известным ему именем и записывает в него свой идентификатор. Сервер выходит из цикла, прочитав идентификатор клиента, и печатает его.

```
      /* процесс-сервер*/
      #include <stdio.h>
      #include <sys/types.h>
      #include <sys/stat.h>
      #include <sys/file.h>
int main(int argc, char **argv)
      {
            int fd;
            int pid;
            mkfifo("fifo", S_IFIFO | 0666);
            /*создали специальный файл FIFO с открытыми для всех
              правами доступа на чтение и запись*/
            fd = open ("fifo", O_RDONLY | O_NONBLOCK);
            /* открыли канал на чтение*/
            while  ( read (fd, &pid, sizeof(int) ) == -1) ;
            printf ("Server %d  got message from %d !\n", getpid(), pid);
            close (fd);
            unlink ("fifo");/*уничтожили именованный канал*/

      }

      /* процесс-клиент*/
      #include <stdio.h>
      #include <sys/types.h>
      #include <sys/stat.h>
      #include <sys/file.h>
int main(int argc, char **argv)
      {
            int fd;
            int pid=getpid( );
            fd = open ("fifo", O_RDWR);
            write (fd, &pid, sizeof(int));
            close (fd);
}
```

6.2.4 Взаимодействие процессов по модели «главный-подчиненный».

Обзор форм межпроцессного взаимодействия в UNIX был бы не полон, если бы мы не рассмотрели простейшую форму взаимодействия, используемую для отладки — трассировку процессов. Принципиальное отличие трассировки от остальных видов межпроцессного взаимодействия в том, что она реализует модель «главный-подчиненный»: один процесс получает возможность управлять ходом выполнения, а также данными и кодом другого.
В UNIX трассировка возможна только между родственными процессами: процесс-родитель может вести трассировку только непосредственно порожденных им потомков, при этом трассировка начинается только после того, как процесс-потомок дает разрешение на это.
Далее схема взаимодействия процессов путем трассировки такова: выполнение отлаживаемого процесса-потомка приостанавливается всякий раз при получении им какого-либо сигнала, а также при выполнении вызова **exec()**. Если в это время отлаживающий процесс осуществляет системный вызов **wait()**, этот вызов немедленно возвращает управление. В то время, как трассируемый процесс находится в приостановленном состоянии, процесс-отладчик имеет возможность анализировать и изменять данные в адресном пространстве отлаживаемого процесса и в пользовательской составляющей его контекста. Далее, процесс-отладчик возобновляет выполнение трассируемого процесса до следующего приостановка (либо, при пошаговом выполнении, для выполнения одной инструкции).

Основной системный вызов, используемый при трассировке,– это **ptrace()**, прототип которого выглядит следующим образом:

```
#include <sys/ptrace.h>
int ptrace(int cmd, pid, addr, data)
```

где **cmd** – код выполняемой команды, **pid –** идентификатор процесса-потомка, **addr** – некоторый адрес в адресном пространстве процесса-потомка, **data** – слово информации.
Чтобы оценить уровень предоставляемых возможностей, рассмотрим основные коды - **cmd** операций этой функции.
cmd = PTRACE_TRACEME — ptrace() с таким кодом операции сыновний процесс вызывает в самом начале своей работы, позволяя тем самым трассировать себя**.** Все остальные обращения к вызову **ptrace()** осуществляет процесс-отладчик.
cmd = PTRACE_PEEKDATA - чтение слова из адресного пространства отлаживаемого процесса по адресу **addr** , **ptrace()** возвращает значение этого слова.
cmd = PTRACE_PEEKUSER — чтение слова из контекста процесса**.** Речь идет о доступе к пользовательской составляющей контекста данного процесса, сгруппированной в некоторую структуру, описанную в заголовочном файле **<sys/user.h>**. В этом случае параметр **addr** указывает смещение относительно начала этой структуры. В этой структуре размещена такая информация, как регистры, текущее состояние процесса, счетчик адреса и так далее. **ptrace()** возвращает значение считанного слова.
cmd = PTRACE_POKEDATA — запись данных, размещенных в параметре **data,** по адресу **addr** в адресном пространстве процесса-потомка.
cmd = PTRACE_POKEUSER — запись слова из **data** в контекст трассируемого процесса со смещением **addr.** Таким образом можно, например, изменить счетчик адреса трассируемого процесса, и при последующем возобновлении трассируемого процесса его выполнение начнется с инструкции, находящейся по заданному адресу.
cmd = PTRACE_GETREGS, PTRACE_GETFREGS — чтение регистров общего назначения (в т.ч. с плавающей точкой) трассируемого процесса и запись их значения по адресу **data**.

cmd = PTRACE_SETREGS, PTRACE_SETFREGS — запись в регистры общего назначения (в т.ч. с плавающей точкой) трассируемого процесса данных, расположенных по адресу **data** в трассирующем процессе.
cmd = PTRACE_CONT — возобновление выполнения трассируемого процесса**.** Отлаживаемый процесс будет выполняться до тех пор, пока не получит какой-либо сигнал, либо пока не завершится.
cmd = PTRACE_SYSCALL, PTRACE_SINGLESTEP — эта команда, аналогично **PTRACE_CONT**, возобновляет выполнение трассируемой программы, но при этом произойдет ее остановка после того, как выполнится одна инструкция. Таким образом, используя **PTRACE_SINGLESTEP**, можно организовать пошаговую отладку. С помощью команды **PTRACE_SYSCALL** возобновляется выполнение трассируемой программы вплоть до ближайшего входа или выхода из системного вызова. Идея использования **PTRACE_SYSCALL** в том, чтобы иметь возможность контролировать значения аргументов, переданных в системный вызов трассируемым процессом, и возвращаемое значение, переданное ему из системного вызова.
cmd = PTRACE_KILL — завершение выполнения трассируемого процесса.

Пример. Общая схема использования механизма трассировки.

Рассмотрим некоторый модельный пример, демонстрирующий общую схему построения отладочной программы (см. также Рис. 7):

```
      ...
if ((pid = fork()) == 0)
{
      ptrace(PTRACE_TRACEME, 0, 0, 0);
      /* сыновний процесс разрешает трассировать себя */
      exec("трассируемый процесс", 0);
      /* замещается телом процесса, который необходимо трассировать */
}
while (1)
{
      /* это процесс, управляющий трассировкой */
      wait((int ) 0);
            /*  процесс  приостанавливается  до  тех  пор,  пока  от
            трассируемого процесса не придет сообщение о том, что он
            приостановился */
      ptrace(cmd, pid, addr, data);
      /* теперь выполняются любые действия над трассируемым
      процессом */
      …
      ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
      /* возобновляем выполнение трассируемой программы */
}
```

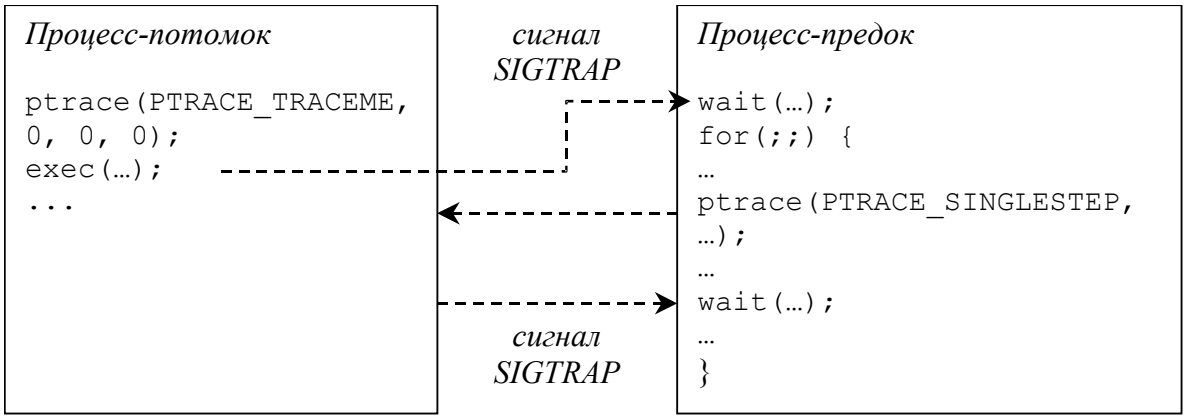


Рис. 7 Общая схема трассировки процессов

Предназначение процесса-потомка — разрешить трассировку себя. После вызова **ptrace(PTRACE_TRACEME, 0, 0, 0)** ядро устанавливает для этого процесса бит трассировки. Сразу же после этого можно заместить код процесса-потомка кодом программы, которую необходимо отладить. Отметим, что при выполнении системного вызова **exec()**, если для данного процесса ранее был установлен бит трассировки, ядро перед передачей управления в новую программу посылает процессу сигнал **SIGTRAP**. При получении данного сигнала трассируемый процесс приостанавливается, и ядро передает управление процессу-отладчику, выводя его из ожидания в вызове **wait().**

Процесс-родитель вызывает **wait()** и переходит в состояние ожидания до того момента, пока потомок не перейдет в состояние трассировки. Проснувшись, управляющий процесс, выполняя функцию **ptrace(cmd, pid, addr, data)** с различными кодами операций, может производить любое действие с трассируемой программой, в частности, читать и записывать данные в адресном пространстве трассируемого процесса, производить его пошаговое выполнение – при этом, как показано в примере выше, применяется та же схема: процесс-отладчик вызывает **wait()** в состояние ожидания, а ядро возобновляет выполнение трассируемого потомка, исполняет трассируемую команду, и вновь передает управление отладчику, выводя его из ожидания .

Пример.

```
/* Процесс-сын: */
int main(int argc, char **argv)
{     /* деление на ноль – здесь процессу будет послан сигнал SIGFPE –
       floating point exception */
      return argc/0;
}
/*Процесс-родитель:*/
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/user.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{    pid_t pid;
      int status;
      struct user_regs_struct REG;
      if  ((pid = fork()) == 0) {
            /*находимся в процессе-потомке, разрешаем трассировку */
            ptrace(PTRACE_TRACEME, 0, 0, 0);
            execl("son", "son", 0);      /* замещаем тело процесса */
                  /* здесь процесс-потомок будет остановлен с сигналом
                  SIG_TRAP, ожидая команды продолжения выполнения от
                  управляющего процесса*/
      }
      /* в процессе-родителе */
      while (1) {
            /* ждем, когда отлаживаемый процесс приостановится */
            wait( &status );
            /*читаем содержимое регистров отлаживаемого процесса */
            ptrace(PTRACE_GETREGS, pid, &REG, &REG);
            /* выводим статус отлаживаемого процесса, номер сигнала,
            который его остановил и значения прочитанных регистров  */
            printf("signal = %d, status = %#x, EIP=%#x ESP=%#x\n",
                  WSTOPSIG(status), status, REG.eip, REG.esp);
            if (WSTOPSIG(status) != SIGTRAP) {
                  if (!WIFEXITED(status)) {
                  /* завершаем выполнение трассируемого процесса */
                        ptrace (PTRACE_KILL, pid, 0, 0);
                  }
                  break;
            }
            /* разрешаем выполнение трассируемому процессу */
            ptrace (PTRACE_CONT, pid, 0, 0);
      }
}
```

6.3 Система межпроцессного взаимодействия IPC.

6.3.1 Общие концепции

Для всех средств IPC приняты общие правила именования объектов, позволяющие процессу получить доступ к такому объекту. Для именования объекта IPC используется ключ, представляющий собой целое число. Ключи являются уникальными во всей UNIX-системе идентификаторами объектов IPC, и зная ключ для некоторого объекта, процесс может получить к нему доступ. При этом процессу возвращается дескриптор объекта, который в дальнейшем используется для всех операций с ним. Проведя аналогию с файловой системой, можно сказать, что ключ аналогичен имени файла, а получаемый по ключу дескриптор – файловому дескриптору, получаемому во время операции открытия файла. Ключ для каждого объекта IPC задается в момент его создания тем процессом, который его порождает, а все процессы, желающие получить в дальнейшем доступ к этому объекту, должны указывать тот же самый ключ.

Итак, все процессы, которые хотят работать с одним и тем же IPC-ресурсом, должны знать некий целочисленный ключ, по которому можно получить к нему доступ. В принципе, программист, пишущий программы для работы с разделяемым ресурсом, может просто жестко указать в программе некоторое константное значение ключа для именования разделяемого ресурса. Однако, возможна ситуация, когда к моменту запуска такой программы в системе уже существует разделяемый ресурс с таким значением ключа, и в виду того, что ключи должны быть уникальными во всей системе, попытка породить второй ресурс с таким же ключом закончится неудачей (подробнее этот момент будет рассмотрен ниже).

Генерация ключей: функция ftok().

Необходим механизм уникального именования ресурса, но вместе с тем нужно, чтобы этот механизм позволял всем процессам, желающим работать с одним ресурсом, получить одно и то же значение ключа.

Для решения этой задачи служит функция **ftok()**:

```
#include <sys/types.h>
#include<sys/ipc.h>
      key_t ftok (char *filename, char proj)
```

Эта функция генерирует значение ключа по некоторой строке символов и добавочному символу, передаваемым в качестве параметров. Гарантируется, что полученное таким образом значение будет отличаться от всех других значений, сгенерированных функцией **ftok()** с другими значениями параметров, и в то же время, при повторном запуске **ftok()** с теми же параметрами, будет получено то же самое значение ключа.
Смысл второго аргумента функции **ftok()** – добавочного символа – в том, что он позволяет генерировать разные значения ключа по одному и тому же значению первого параметра – строки. Это позволяет программисту поддерживать несколько версий своей программы, которые будут использовать одну и ту же строку, но разные добавочные символы для генерации ключа, и тем самым получат возможность в рамках одной системы работать с разными разделяемыми ресурсами.
Следует заметить, что функция **ftok()** не является системным вызовом, а предоставляется библиотекой.

Общие принципы работы с разделяемыми ресурсами.

Рассмотрим некоторые моменты, общие для работы со всеми разделяемыми ресурсами IPC. Как уже говорилось, общим для всех ресурсов является механизм именования. Кроме того, для каждого IPC-ресурса поддерживается идентификатор его владельца и структура, описывающая права доступа к нему. Подобно файлам, права доступа задаются отдельно для владельца, его группы и всех остальных пользователей; однако, в отличие от файлов, для разделяемых ресурсов поддерживается только две категории доступа: по чтению и записи. Априори считается, что возможность изменять свойства ресурса и удалять его имеется только у процесса, эффективный идентификатор пользователя которого совпадает с идентификатором владельца ресурса. Владельцем ресурса назначается пользователь, от имени которого выполнялся процесс, создавший ресурс, однако создатель может передать права владельца другому пользователю. В файле **<sys/ipc.h>** определен тип **struct ipc_perm**, который описывает права доступа к любому IPC-ресурсу. Поля в этой структуре содержат информацию о создателе и владельце ресурса и их группах, правах доступа к ресурсу и его ключе.

Для создания разделяемого ресурса с заданным ключом, либо подключения к уже существующему ресурсу с таким ключом используются ряд системных вызовов, имеющих общий суффикс **get**. Общими параметрами для всех этих вызовов являются ключ и флаги. В качестве значения ключа при создании любого IPC-объекта может быть указано значение **IPC_PRIVATE**. При этом создается ресурс, который будет доступен только породившему его процессу. Такие ресурсы обычно порождаются родительским процессом, который затем сохраняет полученный дескриптор в некоторой переменной и порождает своих потомков. Т.к. потомкам доступен уже готовый дескриптор созданного объекта, они могут непосредственно работать с ним, не обращаясь предварительно к «**get»-**методу. Таким образом, созданный ресурс может совместно использоваться родительским и порожденными процессами. Однако, важно понимать, что если один из этих процессов повторно вызовет «**get»-**метод с ключом **IPC_PRIVATE**, в результате будет получен другой, совершенно новый разделяемый ресурс, т.к. при обращении к «**get»-**методу с ключом **IPC_PRIVATE** всякий раз создается новый объект нужного типа.

Если при обращении к «**get»-**методу указан ключ, отличный от **IPC_PRIVATE**, происходит следующее:

- Происходит поиск объекта с заданным ключом среди уже существующих объектов нужного типа. Если объект с указанным ключом не найден, и среди флагов указан флаг **IPC_CREAT,** будет создан новый объект. При этом значение параметра флагов должно содержать побитовое сложение флага **IPC_CREAT** и константы, указывающей права доступа для вновь создаваемого объекта.
- Если объект с заданным ключом не найден, и среди переданных флагов отсутствует флаг **IPC_CREAT**, «**get»-**метод вернет –1, а в переменной **errno** будет установлено значение **ENOENT**
- Если объект с заданным ключом найден среди существующих, «**get»-**метод вернет дескриптор для этого существующего объекта, т.е. фактически, в этом случае происходит подключение к уже существующему объекту по заданному ключу. Если процесс ожидал создания нового объекта по указанному ключу, то для него такое поведение может оказаться нежелательным, т.к. это будет означать, что в результате случайного совпадения ключей (например, если процесс не использовал функцию **ftok()**) он подключился к чужому ресурсу. Чтобы избежать такой ситуации, следует указать в параметре флагов наряду с флагом **IPC_CREAT** и правами доступа еще и флаг **IPC_EXCL –** в этом случае «**get»-**метод вернет -1, если объект с таким ключом уже существует (переменная **errno** будет установлена в значение **EEXIST**)

- Следует отметить, что при подключении к уже существующему объекту дополнительно проверяются права доступа к нему. В случае, если процесс, запросивший доступ к объекту, не имеет на то прав, «**get**»-метод вернет –1, а в переменной **errno** будет установлено значение **EACCESS**

Нужно заметить, что для каждого типа объектов IPC существует некое ограничение на максимально возможное количество одновременно существующих в системе объектов данного типа. Если при попытке создания нового объекта окажется, что указанное ограничение превышено, «**get**»-метод, совершавший попытку создания объекта, вернет -1, а в переменной **errno** будет указано значение **ENOSPC**.

Отметим, что даже если ни один процесс не подключен к разделяемому ресурсу, система не удаляет его автоматически. Удаление объектов IPC является обязанностью одного из работающих с ним процессов и для этого определена специальная функция. Для этого системой предоставляются соответствующие функции по управлению объектами System V IPC.

6.3.2 IPC: очередь сообщений.

Итак, одним из типов объектов System V IPC являются очереди сообщений. Очередь сообщений представляет собой некое хранилище типизированных сообщений, организованное по принципу FIFO. Любой процесс может помещать новые сообщения в очередь и извлекать из очереди имеющиеся там сообщения. Каждое сообщение имеет тип, представляющий собой некоторое целое число. Благодаря наличию типов сообщений, очередь можно интерпретировать двояко — рассматривать ее либо как сквозную очередь неразличимых по типу сообщений, либо как некоторое объединение подочередей, каждая из которых содержит элементы определенного типа. Извлечение сообщений из очереди происходит согласно принципу FIFO – в порядке их записи, однако процесс-получатель может указать, из какой подочереди он хочет извлечь сообщение, или, иначе говоря, сообщение какого типа он желает получить – в этом случае из очереди будет извлечено самое «старое» сообщение нужного типа.
Рассмотрим набор системных вызовов, поддерживающий работу с очередями сообщений.

Доступ к очереди сообщений.

Для создания новой или для доступа к существующей используется системный вызов:

```
#INCLUDE <SYS/TYPES.H>
#INCLUDE <SYS/IPC.H>
#INCLUDE <SYS/MESSAGE.H>
INT MSGGET (KEY_T KEY, INT MSGFLAG)
```

В случае успеха вызов возвращает положительный дескриптор очереди, который может в дальнейшем использоваться для операций с ней, в случае неудачи -1. Первым аргументом вызова является ключ, вторым – флаги, управляющие поведением вызова. Подробнее детали процесса создания/подключения к ресурсу описаны выше.

Отправка сообщения.

Для отправки сообщения используется функция **msgsnd():**

```
#INCLUDE <SYS/TYPES.H>
#INCLUDE <SYS/IPC.H>
#INCLUDE <SYS/MSG.H>
INT MSGSND (INT MSQID, CONST VOID *MSGP, SIZE_T MSGSZ, INT MSGFLG)
```

Ее первый аргумент — идентификатор очереди, полученный в результате вызова **msgget()**. Второй аргумент — указатель на буфер, содержащий реальные данные и тип сообщения, подлежащего посылке в очередь, в третьем аргументе указывается размер буфера.
В качестве буфера необходимо указывать структуру, содержащую следующие поля (в указанном порядке):
long msgtype — тип сообщения
char msgtext[ ] — данные (тело сообщения)
В заголовочном файле **<sys/msg.h>** определена константа **MSGMAX**, описывающая максимальный размер тела сообщения. При попытке отправить сообщение, у которого число элементов в массиве **msgtext** превышает это значение, системный вызов вернет –1.
Четвертый аргумент данного вызова может принимать значения 0 или **IPC_NOWAIT**. В случае отсутствия флага **IPC_NOWAIT** вызывающий процесс будет блокирован (т.е. приостановит работу), если для посылки сообщения недостаточно системных ресурсов, т.е. если полная длина сообщений в очереди будет больше максимально допустимого. Если же флаг **IPC_NOWAIT** будет установлен, то в такой ситуации выход из вызова произойдет немедленно, и возвращаемое значение будет равно –1.
В случае удачной записи возвращаемое значение вызова равно 0.

Получение сообщения.

Для получения сообщения имеется функция **msgrcv:**

```
#INCLUDE <SYS/TYPES.H>
#INCLUDE <SYS/IPC.H>
#INCLUDE <SYS/MSG.H>
INT MSGRCV (INT MSQID, VOID *MSGP, SIZE_T MSGSZ, LONG MSGTYP, INT MSGFLG)
```

Первые три аргумента аналогичны аргументам предыдущего вызова: это дескриптор очереди, указатель на буфер, куда следует поместить данные, и максимальный размер (в байтах) тела сообщения, которое можно туда поместить. Буфер, используемый для приема сообщения, должен иметь структуру, описанную выше.
Четвертый аргумент указывает тип сообщения, которое процесс желает получить. Если значение этого аргумента есть 0, то будет получено сообщение любого типа. Если значение аргумента **msgtyp** больше 0, из очереди будет извлечено сообщение указанного типа. Если же значение аргумента **msgtyp** отрицательно, то тип принимаемого сообщения определяется как наименьшее значение среди типов, которые меньше модуля **msgtyp**. В любом случае, как уже говорилось, из подочереди с заданным типом (или из общей очереди, если тип не задан) будет выбрано самое старое сообщение.
Последним аргументом является комбинация (побитовое сложение) флагов. Если среди флагов не указан **IPC_NOWAIT**, и в очереди не найдено ни одного сообщения, удовлетворяющего критериям выбора, процесс будет заблокирован до появления такого сообщения. (Однако, если такое сообщение существует, но его длина превышает указанную в аргументе **msgsz**, то процесс заблокирован не будет, и вызов сразу вернет –1. Сообщение при этом останется в очереди). Если же флаг **IPC_NOWAIT** указан, то вызов сразу вернет –1.
Процесс может также указать флаг **MSG_NOERROR** – в этом случае он может прочитать сообщение, даже если его длина превышает указанную емкость буфера. В этом случае в буфер будет записано первые **msgsz** байт из тела сообщения, а остальные данные отбрасываются.
В случае удачного чтения возвращаемое значение вызова равно 0.

Управление очередью сообщений.

Функция управления очередью сообщений выглядит следующим образом:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msgid_ds *buf)
```

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с очередью и уничтожения очереди. Ее аргументы — идентификатор ресурса, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры очереди. Тип **msgid_ds** описан в заголовочном файле **<sys/message.h>**, и представляет собой структуру, в полях которой хранятся права доступа к очереди, статистика обращений к очереди, ее размер и т.п.

Возможные значения аргумента **cmd**:

IPC_STAT – скопировать структуру, описывающую управляющие параметры очереди по адресу, указанному в параметре **buf**

IPC_SET – заменить структуру, описывающую управляющие параметры очереди, на структуру, находящуюся по адресу, указанному в параметре **buf**

IPC_RMID – удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

Пример. **Использование очереди сообщений.**

Пример программы, где основной процесс читает некоторую текстовую строку из стандартного ввода и в случае, если строка начинается с буквы **'a'**, то эта строка в качестве сообщения будет передана процессу **A**, если **'b'** - процессу **B,** если **'q'** - то процессам **A** и **B**, затем будет осуществлен выход. Процессы **A** и **B** распечатывают полученные строки на стандартный вывод.

6.3.2.1.1.1.1.1.1 Основной процесс.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
#include <stdio.h>
struct {long mtype;                /* тип сообщения */
      char Data[256];              /* сообщение */
           } Message;
int main(int argc, chr **argv)
{     key_t key;  int msgid;  char str[256];
      key=ftok("/usr/mash",'s');
      /*получаем уникальный ключ, однозначно определяющий
      доступ к ресурсу данного типа */
      msgid=msgget(key, 0666 | IPC_CREAT);
      /*создаем очередь сообщений , 0666 определяет права доступа */
      for(;;)
      {/* запускаем вечный цикл */
            gets(str); /* читаем из стандартного ввода строку */
            strcpy(Message.Data, str);
            /* и копируем ее в буфер сообщения */
            switch(str[0]){
                  case 'a':
                  case 'A':
                        Message.mtype=1;  /* устанавливаем тип*/
                        msgsnd(msgid, (struct msgbuf*) (&Message),
                                          strlen(str)+1, 0);
                        /* посылаем сообщение в очередь*/
```

```
                    break;
               case 'b':
               case 'B':
                    Message.mtype=2;
                    msgsnd(msgid, (struct msgbuf*) (&Message),
                                        strlen(str)+1, 0);
                    break;
               case q':
               case 'Q':
                    Message.mtype=1;
                    msgsnd(msgid, (struct msgbuf*) (&Message),
                                         strlen(str)+1, 0);
                    Message.mtype=2;
                    msgsnd(msgid, (struct msgbuf*) (&Message),
                                        strlen(str)+1, 0);
                    sleep(10);
     /* ждем получения сообщений процессами A и B */
                      msgctl(msgid, IPC_RMID, NULL);
                         /* уничтожаем очередь*/
                      exit(0);
               default: break;
               }
     }
}
```

Процесс-приемник A

```
/* процесс B аналогичен с точностью до четвертого параметра
в msgrcv */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
#include <stdio.h>
struct {   long mtype;
     char Data[256];
          } Message;
int main(int argc, chr **argv)
{    key_t key;
     int msgid;
     key=ftok("/usr/mash",'s');
     /* получаем ключ по тем же параметрам */
     msgid=msgget(key, 0666 | IPC_CREAT);
     /*подключаемся к очереди сообщений */
     for(;;) {/* запускаем вечный цикл */
          msgrcv(msgid, (struct msgbuf*) (&Message), 256, 1, 0);
          /* читаем сообщение с типом 1*/
          printf("%s",Message.Data);

          if (Message.Data[0]='q' || Message.Data[0]='Q') break;
          }
     exit();
}
```

Благодаря наличию типизации сообщений, очередь сообщений предоставляет возможность мультиплексировать сообщения от различных процессов, при этом каждая пара взаимодействующих через очередь процессов может использовать свой тип сообщений, и таким образом, их данные не будут смешиваться.

В качестве иллюстрации приведем следующий стандартный пример взаимодействия.

Рассмотрим еще один пример - пусть существует **процесс-сервер** и несколько **процессов-клиентов**. Все они могут обмениваться данными, используя одну очередь сообщений. Для этого сообщениям, направляемым от клиента к серверу, присваиваем значение типа **1**. При этом процесс, отправивший сообщение, в его теле передает некоторую информацию, позволяющую его однозначно идентифицировать. Тогда сервер, отправляя сообщение конкретному процессу, в качестве его типа указывает эту информацию (например, **PID** процесса). Таким образом, сервер будет читать из очереди только сообщения типа **1**, а клиенты — сообщения с типами, равными идентификаторам их процессов.

Пример. Очередь сообщений.
Модель «клиент-сервер»

```
/* server */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, chr **argv)
     {
          struct {
               long mestype;
               char mes [100];
               } messageto;
          struct {
              long mestype;
              long mes;
              } messagefrom;

          key_t key;
          int mesid;
          key=ftok("example",'r');
          mesid=msgget (key, 0666 | IPC_CREAT);
          while(1)
          {
               if (msgrcv(mesid, &messagefrom, sizeof(messagefrom), 1,
          0)<=0) continue;
               messageto.mestype=messagefrom.mes;
               strcpy( messageto.mes, "Message for client");
               msgsnd (mesid, &messageto, sizeof(messageto), 0);
          }
          msgctl (mesid, IPC_RMID, 0);
          return 0;
     }
```

```
 /* client */
 #include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/msg.h>

int main(int argc, chr **argv)
{
struct {
long mestype;   /*описание структуры сообщения*/
long mes;
     } messageto;
struct {
     long mestype;   /*описание структуры сообщения*/
     char mes[100];
     } messagefrom;
key_t key;
int mesid;
long pid=getpid();
key=ftok("example",'r');
mesid=msgget (key,0 );   /*присоединение к очереди сообщений*/
messageto.mestype=1;
messageto.mes=pid;
msgsnd (mesid, &messageto, sizeof(messageto), 0); /*посылка */
while ( msgrcv (mesid, &messagefrom, sizeof(messagefrom), pid,0) <=0);
/*прием собщения */
printf("%s",messagefrom.mes);
return 0;
}
```

6.3.3 IPC: разделяемая память.

Механизм разделяемой памяти позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти. Благодаря этому, данные, находящиеся в этой области памяти, будут доступны для чтения и модификации всем процессам, подключившимся к данной области памяти.
Процесс, подключившийся к разделяемой памяти, может затем получить указатель на некоторый адрес в своем виртуальном адресном пространстве, соответствующий данной области разделяемой памяти. После этого он может работать с этой областью памяти аналогично тому, как если бы она была выделена динамически (например, путем обращения к malloc()), однако, как уже говорилось, разделяемая область памяти не уничтожается автоматически даже после того, как процесс, создавший или использовавший ее, перестанет с ней работать.
Рассмотрим набор системных вызовов для работы с разделяемой памятью.

Создание общей памяти.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

      int shmget (key_t key, int size, int shmemflg)
```

Аргументы этого вызова: **key -** ключ для доступа к разделяемой памяти; **size** задает размер области памяти, к которой процесс желает получить доступ. Если в результате вызова **shmget()** будет создана новая область разделяемой памяти, то ее размер будет соответствовать значению **size**. Если же процесс подключается к существующей области разделяемой памяти, то значение **size** должно быть не более ее размера, иначе вызов вернет – 1. Заметим, что если процесс при подключении к существующей области разделяемой памяти указал в аргументе **size** значение, меньшее ее фактического размера, то впоследствии он сможет получить доступ только к первым **size** байтам этой области.
Третий параметр определяет флаги, управляющие поведением вызова. Подробнее алгоритм создания/подключения разделяемого ресурса был описан выше.
В случае успешного завершения вызов возвращает положительное число – дескриптор области памяти, в случае неудачи - -1.

Доступ к разделяемой памяти.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

      char *shmat(int shmid, char *shmaddr, int shmflg)
```

При помощи этого вызова процесс подсоединяет область разделяемой памяти, дескриптор которой указан в **shmid**, к своему виртуальному адресному пространству. После выполнения этой операции процесс сможет читать и модифицировать данные, находящиеся в области разделяемой памяти, адресуя ее как любую другую область в своем собственном виртуальном адресном пространстве.
В качестве второго аргумента процесс может указать виртуальный адрес в своем адресном пространстве, начиная с которого необходимо подсоединить разделяемую память. Чаще всего, однако, в качестве значения этого аргумента передается 0, что означает, что система

сама может выбрать адрес начала разделяемой памяти. Передача конкретного адреса в этом параметре имеет смысл в том случае, если, к примеру, в разделяемую память записываются указатели на нее же (например, в ней хранится связанный список) – в этой ситуации для того, чтобы использование этих указателей имело смысл и было корректным для всех процессов, подключенных к памяти, важно, чтобы во всех процессах адрес начала области разделяемой памяти совпадал.
Третий аргумент представляет собой комбинацию флагов. В качестве значения этого аргумента может быть указан флаг **SHM_RDONLY**, который указывает на то, что подсоединяемая область будет использоваться только для чтения.
Эта функция возвращает адрес, начиная с которого будет отображаться присоединяемая разделяемая память. В случае неудачи вызов возвращает **-1**.

Открепление разделяемой памяти.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

      int shmdt(char *shmaddr)
```

Данный вызов позволяет отсоединить разделяемую память, ранее присоединенную посредством вызова **shmat()**
shmaddr - адрес прикрепленной к процессу памяти, который был получен при вызове **shmat()**
В случае успешного выполнения функция возвращает 0, в случае неудачи -1

Управление разделяемой памятью.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
      int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложения и снятия блокировки на нее и ее уничтожения. Аргументы вызова — дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти. Тип **shmid_ds** описан в заголовочном файле **<sys/shm.h>**, и представляет собой структуру, в полях которой хранятся права доступа к области памяти, ее размер, число процессов, подсоединенных к ней в данный момент, и статистика обращений к области памяти.
Возможные значения аргумента **cmd**:
IPC_STAT – скопировать структуру, описывающую управляющие параметры области памяти по адресу, указанному в параметре **buf**
IPC_SET – заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре **buf**. Выполнить эту операцию может процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя, при этом процесс может изменить только владельца области памяти и права доступа к ней.
IPC_RMID – удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

SHM_LOCK, SHM_UNLOCK – блокировать или разблокировать область памяти. Выполнить эту операцию может только процесс с правами привилегированного пользователя.

Пример. Работа с общей памятью в рамках одного процесса.

```
int main(int argc, chr **argv)
{
      key_t key;
      char *shmaddr;
      key=ftok("/tmp/ter",'S');
      shmid=shmget(key, 100,0666|IPC_CREAT);
      shmaddr=shmat(shmid,NULL,0); /*подключение к памяти*/
      putm(shmaddr); /*работа с ресурсом*/
      waitprocess();
      shmctl(shmid,IPC_RMID,NULL); /*уничтожение  ресурса*/
      exit();
}
```

6.3.4 IPC: массив семафоров.

Семафоры представляют собой одну из форм IPC и используются для синхронизации доступа нескольких процессов к разделяемым ресурсам, т.е. фактически они разрешают или запрещают процессу использование разделяемого ресурса. В начале излагалась идея использования такого механизма. Речь шла о том, что при наличии некоторого разделяемого ресурса , с которым один из процессов работает, необходимо блокировать доступ к нему других процессов. Для этого с ресурсом связывается некоторая переменная-счетчик, доступная для всех процессов. При этом считаем, что значение счетчика, равное 1 будет означать доступность ресурса, а значение, равное 0 — его занятость. Далее работа организуется следующим образом: процесс, которому необходим доступ к файлу, проверяет значение счетчика, если оно равно 0, то он в цикле ожидает освобождения ресурса, если же оно равно 1, процесс устанавливает значение счетчика равным 0 и работает с ресурсом. После завершения работы необходимо открыть доступ к ресурсу другим процессам, поэтому снова сбрасывается значение счетчика на 1. В данном примере счетчик и есть **семафор.**
Семафор находится адресном пространстве ядра и все операции выполняются также в режиме ядра.
В System V IPC семафор представляет собой группу (вектор) счетчиков, значения которых могут быть произвольными в пределах, определенных системой (не только 0 и 1).

Доступ к семафору

Для получения доступа (или его создания) к семафору используется системный вызов:

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semget (key_t key, int nsems, int semflag).
```

Первый параметр функции **semget()** - ключ, второй - количество семафоров (длина массива семафоров) и третий параметр - флаги. Через флаги можно определить права доступа и те операции, которые должны выполняться (открытие семафора, проверка, и т.д.). Функция **semget()** возвращает целочисленный идентификатор созданного разделяемого ресурса, либо -1, если ресурс не удалось создать.

Операции над семафором

С полученным идентификатором созданного объекта можно производить операции с семафором, для чего используется системный вызов **semop():**

```
int semop (int semid, struct sembuf *semop, size_t nops)
```

Первый аргумент – идентификатор ресурса, второй аргумент является указателем на структуру, определяющую операции, которые необходимо произвести над семафором. Третий параметр - количество указателей на эту структуру, которые передаются функцией **semop()**. То есть операций может быть несколько и операционная система гарантирует их атомарное выполнение.

Структура имеет **sembuf** вид:

```
struct sembuf { short sem_num;    /*номер семафора в векторе*/
                short sem_op;  /*производимая операция*/
                short sem_flg;      /*флаги операции*/
}
```

Поле операции интерпретируется следующим образом. Пусть значение семафора с номером **sem_num** равно **sem_val**. В этом случае, если значение операции не равно нулю, то оценивается значение суммы **sem_val + sem_op**. Если эта сумма больше либо равна нулю, то значение данного семафора устанавливается равным сумме предыдущего значения и кода операции, т.е. **sem_val:= sem_val+sem_op.** Если эта сумма меньше нуля, то действие процесса будет приостановлено до наступления одного из следующих событий:

1. Значение суммы **sem_val + sem_op** станет больше либо равно нулю.
2. Пришел какой-то сигнал. Значение **semop** в этом случае будет равно -1.

Если код операции **semop** равен нулю, то процесс будет ожидать обнуления семафора. Если мы обратились к функции **semop** с нулевым кодом операции, а к этому моменту значение семафора стало равным нулю, то никакого ожидания не происходит.

Рассмотрим третий параметр - флаги. Если третий параметр равен нулю, то это означает, что флаги не используются. Флагов имеется большое количество в т.ч. **IPC_NOWAIT** (при этом флаге во всех тех случаях, когда мы говорили, что процесс будет ожидать, он не будет ожидать).

6.3.5 IPC: Пример. Использование разделяемой памяти и семафоров.

Рассмотрим двухпроцессную программу:

1 процесс - создает ресурсы “разделяемая память” и “семафоры”, далее он начинает принимать строки со стандартного ввода и записывает их в разделяемую память.

2 процесс - читает строки из разделяемой памяти.

1й процесс:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX    256
int main(int argc, char **argv)
{
      key_t key;
      int semid, shmid;
      struct sembuf sops;
      char *shmaddr;
      char str[NMAX];

      key = ftok(“/usr/ter/exmpl”,’S’);
      /* создаем уникальный ключ */
      semid = semget(key, 1, 0666 | IPC_CREAT);
      /* создаем один семафор с определенными правами доступа */
      shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
      /*создаем разделяемую память на 256 элементов */
      shmaddr = shmat(shmid, NULL, 0);
            /*подключаемся к разделу памяти, в shaddr –
              указатель на буфер с разделяемой памятью*/
      semctl(semid,0,SETVAL, (int) 0);
      /*инициализируем семафор значением 0 */
      sops.sem_num = 0;
      sops.sem_flg = 0;
      do { /* запуск цикла */
            printf(“Введите строку:”);
            if (fgets(str, NMAX, stdin) == NULL) /* окончание ввода */
                  /* пишем признак завершения – строку “Q” */
                  strcpy(str, “Q”);

            /* в текущий момент семафор открыт для этого процесса*/
            strcpy(shmaddr, str); /* копируем строку в разд. память */
            /* предоставляем второму процессу возможность войти */
            sops.sem_op=3; /* увеличение семафора на 3 */
            semop(semid, &sops, 1);
            /* ждем, пока семафор будет открыт для 1го процесса -
               для следующей итерации цикла*/
            sops.sem_op=0; /* ожидание обнуления семафора */
            semop(semid, &sops, 1);
      } while (str[0] != ‘Q’);
      /* в данный момент второй процесс уже дочитал из разделяемой памяти
и отключился от нее – можно ее удалять*/
```

```
      shmdt(shmaddr) ; /* отключаемся от разделяемой памяти */
      shmctl(shmid, IPC_RMID, NULL);
      /* уничтожаем разделяемую память */
      semctl(semid, 0, IPC_RMID, (int) 0);
      /* уничтожаем семафор */
      return 0;
}
```

2й процесс:

```
/* необходимо корректно определить существование ресурса, если он есть -
подключиться */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX    256

int main(int argc, char **argv)
{
      key_t key;
      int semid, shmid;
      struct sembuf sops;
      char *shmaddr;
      char str[NMAX];

      key = ftok("/usr/ter/exmpl",'S');
      /* создаем тот же самый ключ */
      semid = semget(key, 1, 0666 | IPC_CREAT);
      shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
      /* аналогично предыдущему процессу  - инициализации ресурсов */
      shmaddr = shmat(shmid, NULL, 0);
      sops.sem_num = 0;
      sops.sem_flg = 0;
      /* запускаем цикл */
      do {  printf("Waiting… \n"); /* ожидание на семафоре */
            sops.sem_op=-2;
                  /* будем ожидать, пока "значение семафора" + "значение
                  sem_op" не станет положительным, т.е. пока значение
                  семафора не станет как минимум 3 (3-2=1 > 0) */
            semop(semid, &sops, 1);
            /* теперь значение семафора равно 1 */
            strcpy(str, shmaddr); /* копируем строку из разд.памяти */
                  /*критическая секция - работа с разделяемой памятью - в
                        этот момент  первый процесс к разделяемой памяти
                        доступа не имеет*/
                  if (str[0] == 'Q')
                        /*завершение работы - освобождаем разделяемую
                        память */
                        shmdt(shmaddr);
            /*после работы – обнулим семафор*/
            sops.sem_op=-1;
            semop(semid, &sops, 1);
            printf("Read from shared memory: %s\n", str);
      } while (str[0] != 'Q');
      return 0;
}
```

Отметим, что данный пример демонстрирует два разных приема использования семафоров для синхронизации: первый процесс блокируется в ожидании обнуления семафора, т.е. для того, чтобы он мог войти в критическую секцию, значение семафора должно стать нулевым; второй процесс блокируется при попытке уменьшить значение семафора до отрицательной величины, для того, чтобы этот процесс мог войти в критическую секцию, значение семафора должно быть не менее 3. Обратите внимание, что в данном примере, помимо взаимного исключения процессов, достигается строгая последовательность действий двух процессов: они получают доступ к критической секции строго по очереди.

7 ОС Unix: Работа с внешними устройствами

Организацию работы системы с внешними устройствами можно рассматривать с различных точек зрения.

Одна – точка зрения пользователя. В этом случае организация работы с внешними устройствами представляется набором интерфейсов и характеристик использования системных вызовов, предоставляемых системой при создании программ. Следует отметить, что в ОС Unix интерфейсы системных вызовов, обеспечивающих работу с внешними устройствами стандартизированы и представляют собой стандартные средства работы с содержимым файлов.

Другая – точка зрения системного администратора перед которым время от времени, возможно, возникают проблемы консультирования пользователей, анализ и локализация нештатных ситуаций, возникающих в процессе работы с внешними устройствами, подключение новых устройств.

Третья – точка зрения системного программиста, для которого интересным является внутренняя организация процессов, связанных с работой внешних устройств.

7.1 Файлы устройств, драйверы

С точки зрения внутренней организации системы, как и в подавляющем большинстве других операционных систем, работа с внешними устройствами осуществляется посредством использования иерархии драйверов, которые позволяют организовывать взаимодействие ядра ОС с конкретными устройствами. В системе Unix существует единый интерфейс организации взаимодействия с внешними устройствами, для этих целей используются ***специальные файлы устройств***, размещенные в каталоге **/dev**. Файл устройства позволяет ассоциировать некоторое имя (имя файла устройства) с драйвером того или иного устройства. Следует отметить, что здесь мы несколько замещаем понятие устройства понятием драйвер устройства, так как несмотря на то, что мы используем термин ***специальные файлы устройств***, на практике, мы используем ассоциированный с данным специальным файлом драйвер устройства, и таких драйверов у одного устройства может быть произвольное число. Возможно, более удачным было бы использовать специальный файл-драйвер устройства.

В системе существуют два типа специальных файлов устройств:

- файлы байториентированных устройств (драйверы обеспечивают возможность побайтного обмена данными и, обычно, не используют централизованной внутрисистемной кэш-буферизации);
- файлы блокориентированных устройств (обмен с данными устройствами осуществляется фиксированными блоками данных, обмен осуществляется с использованием специального внутрисистемного буферного кэша).

Следует отметить, файловая система может быть создана только на блокориентированных устройствах.
В общем случае тип файла определяется свойствами конкретного устройства и организацией драйвера. Конкретное физическое устройство может иметь, как байториентированные драйверы драйверы, так и блокориентированные. Например, если рассмотреть физическое устройство *Оперативная память* , для него можно реализовать, как байториентированный интерфейс обмена (и соответствующий байториентированный драйвер), так и блокориентированный.
Содержимое файлов устройств размещается исключительно в соответствующем индексном дескрипторе, структура которого для фалов данного типа, отличается от структуры индексных дескрипторов других типов файлов. Итак индексный дескриптор файла устройства содержит:

- тип файла устройства – байториентированный или блокориентированный;
- «старший номер» (major number) устройства - номер драйвера в соответствующей таблице драйверов устройств;
- «младший номер» (minor number) устройства – служебная информация, передающаяся драйверу устройства.

Система поддерживает две таблицы драйверов устройств.
bdevsw – таблица драйверов блокориентированных устройств. **cdevsw** - таблица байториентированных устройств. Выбор конкретной таблицы определяется типом файла устройства. Соответственно, поле *старший номер* определяет строку таблицы с которой ассоциирован драйвер устройства. Драйверу устройства может быть передана дополнительная информация через поле *младший номер* это может быть, например, номер конкретного однотипного устройства или некоторая информация, определяющая дополнительные функции драйвера. Каждая запись этих таблиц содержит так называемый коммутатор устройства – структуру, в которой размещены указатели на соответствующие точки входа (функции) драйвера. Таким образом, в системе определяется базовый уровень взаимодействия с драйвером устройства (конкретный состав точек входа определяется конкретной версией системы). В случае, если конкретный драйвер устройства не поддерживает работу с той или иной точкой входа, на ее место устанавливается специальная ссылка-заглушка на точку ядра.
В качестве примера, рассмотрим типовой набор точек входа в драйвер (β - префикс точки входа, характеризующий конкретный драйвер):

- **βopen()** открытие устройства, обеспечивается инициализация устройства и внутренних структур данных драйвера;
- **βclose()** закрытие драйвера устройства, например в том случае, если ни один из процессов не работает с драйвером;
- **βread()** чтение данных;
- **βwrite()** запись данных;
- **βioctl()** управление устройством, задание режимов работы драйвера, определение набора внутренних операций/команд драйвера;
- **βintr()** – обработка прерывания, вызывается ядром при возникновении прерывания в устройстве с которым ассоциирован драйвер;
- **βstrategy()** управление стратегией организации блокориентированного обмена (некоторые функции оптимизации организации обмена, обработка специальных ситуаций, связанных с функционированием конкретного устройства и т.п.).

Так в некоторых реализациях системы возможно отсутствие точек входа чтения и записи для блокориентированнх устройств. В этом случае блокориентированный обмен реализуются путем передачи управления на точу **βstrategy().**

В системе возможно обращение к функциям драйвера в следующих ситуациях:

1. старт системы, определение ядром состава доступных устройств.
2. обработка запроса ввода/вывода (запрос может быть инициирован, любыми процессами, в том числе и ядром);
3. обработка прерывания, связанного с данным устройством, в этом случае ядро вызывает специальную функцию драйвера;
4. выполнение специальных команд управления (например, остановка устройства, приведение устройства в некоторое начальное состояние и т.п.).

7.2 Включение/удаление драйверов в систему.

Существует два, традиционных способа включения драйверов новых устройств в систему:

- путем «жесткого», статического встраивания драйвера в код ядра, требующего перекомпиляцию исходных текстов ядра или пересборку объектных модулей ядра.
- за счет динамического включения драйвера в систему.

Динамическое включение драйверов в систему предполагает выполнение следующей последовательности действий:

- загрузка и динамическое связывание драйвера с кодом ядра (выполняется специальным загрузчиком);
- инициализация драйвера и соответствующего ему устройства (создание специальных структур данных драйвера, формирование данных коммутатора устройства, связывание обработчика прерываний ядра с данным драйвером).

Для обеспечения динамического включения/выключения драйверов предоставляется набор системных вызовов, обеспечивающий установку и удаление драйверов в систему.

7.3 Организация обмена данными с файлами.

На практике, наиболее часто мы имеем дело с обменами, связанными с доступом к содержимому обыкновенных файлов. Рассмотрим обобщенную схему организации обмена данными с файлами, т.е. внутреннюю организацию программ и данных, обеспечивающих доступ к содержимому файловой системы (файловая система может быть создана исключительно на блокориентированных устройствах).

Рассмотрим ряд информационных структур и таблиц, используемых системой для организации интерфейса работы с файлами. Операционная система подразделяет данные структура на две категории:

- ассоциированные с процессом;
- ассоциированные с ядром операционной системой.

Таблица индексных дескрипторов открытых файлов.

Для каждого открытого в рамках системы файла формируется запись в таблице ТИДОФ, содержащая:

- копия индексного дескриптора (ИД) открытого файла;
- кратность - счетчик открытых в системе файлов, связанных с данным ИД.

Вся работа с содержимым открытых файлов происходит посредством использования копии ИД, размещенной в таблице ТИДОФ. Данная *таблица размещается в памяти ядра ОС*. Если один и тот же файл открыт неоднократно, то запись в ТИДОФ создается одна, но каждое дополнительное открытие этого файла увеличивает счетчик на единицу

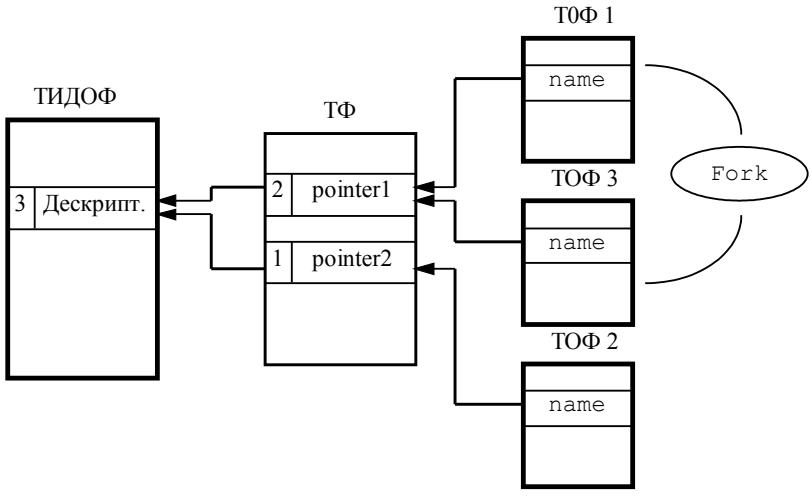
Таблица файлов.

Таблица файлов содержит сведения о всех файловых дескрипторах открытых в системе файлов. Каждая запись ТФ соответствует открытому в системе файлу или точнее используемому файловому дескриптору (ФД). Каждая запись ТФ содержит указатели чтения/записи из/в файл. Рассмотрим правила установления соответствия между открытыми в процессах файлами и записями ТФ. При каждом новом обращении к функции открытия файла в таблице процессов образуется новая запись, таким образом если неоднократно в одном или нескольких процессах открывается один и тот же файл, то в каждом случае будет определяться свой независимый от других файловый дескриптор, в том числе со своим указателем чтения/записи. Если файловый дескриптор в процессе образуется за счет наследования, то в этом случае новые записи в ТФ не образуются, а происходит увеличение счетчика «наследственности» в записи, соответствующей файлу, открытому в прародителе. *Таблица размещается в памяти ОС.*

Таблица открытых файлов.

С каждым процессом связана таблица открытых файлов (ТОФ). Номер записи в данной таблице есть номер ФД, который может использоваться в процессе. Каждая строка этой таблицы имеет ссылку на соответствующую строку ТФ. Первые три строки этой таблицы используются для файловых дескрипторов стандартных устройств/файлов ввода вывода.

Для иллюстрации работы с данными таблицами рассмотрим следующий пример.



Пусть в системе сформирован процесс **№1**, в нем открыт файл с именем **name** (для простоты будем считать, то это единственное открытие файла с данным именем в данный момент времени), в таблице ТОФ№1 этого процесса будет образована соответствующая запись, которая будет ссылаться на запись в ТФ, которая, в свою очередь, ссылается на таблицу ТИДОФ. Счетчик наследственности ТФ и счетчик кратности ТИДОФ будут равны единице. Далее, формируется процесс №2, который в свою очередь открывает файл с именем **name**, в результате чего в ТФ будет образована новая запись, которая будет ссылаться на запись ТИДОФ, соответствующую индексному дескриптору файла name, счетчик кратности этой записи увеличится на единицу.

Процесс №1 выполняет системный вызов **fork()** в результате чего образуется процесс №3 с открытым (унаследованным) файлом **name**. В таблице ТОФ№3 будет размещена копия таблицы ТОФ№2, счетчик наследственности соответствующей записи ТФ и счетчик кратности в записи ТИДОФ увеличатся на единицу.

7.4 Буферизация при блокориентированном обмене.

Особенностью работы с блокориентированными устройствами является возможность организации буферизации при обмене. Суть заключается в следующем. В RAM организуется пул буферов, где каждый буфер имеет размер в один блок. Каждый из этих блоков может быть ассоциирован с драйвером одного из физических блок-ориентированных устройств.

Рассмотрим, как выполняется последовательность действий при исполнении заказа на чтение блока. Будем считать, что поступил заказ на чтение N-ого блока из устройства с номером M.

1. Среди буферов буферного пула осуществляется поиск заданного блока, т.е. если обнаружен буфер, содержащий N-ый блок M-ого устройства, то фиксируем номер этого буфера. В этом случае, обращение к реальному физическому устройству не происходит, а операция чтения информации является представлением информации из найденного буфера. Переходим на шаг 4.
2. Если поиск заданного буфера неудачен, то в буферном пуле осуществляется поиск буфера для чтения и размещения данного блока. Если есть свободный буфер (реально, эта ситуация возможна только при старте системы), то фиксируем его номер и переходим к шагу 3. Если свободного буфера не нашли, то мы выбираем буфер, к которому не было обращений самое долгое время. В случае если в буфере имеется установленный признак произведенной записи информации в буфер, то происходит реальная запись размещенного в буфере блока на физической устройство. Затем фиксируем его номер и также переходим к пункту 3.
3. Осуществляется чтение N-ого блока устройства M в найденный буфер.
4. Происходит обнуление счетчика времени в данном буфере и увеличение на единицу счетчиков в других буферах.
5. Передаем в качестве результата чтения содержимое данного буфера.

Вы видите, что здесь есть оптимизация, связанная с минимизацией реальных обращений к физическому устройству. Это достаточно полезно при работе системы. Запись блоков осуществляется по аналогичной схеме. Таким образом, организована буферизация при низкоуровневом вводе/выводе. Преимущества очевидны. Недостатком является то, что система в этом случае является критичной к несанкционированным отключениям питания, т. е. ситуация, когда буфера системы не выгружены, а происходит нештатное прекращение выполнения программ операционной системы, что может привести к потере информации.

Второй недостаток заключается в том, что за счет буферизации разорваны во времени факт обращения к системе за обменом и реальный обмен. Этот недостаток проявляется в случае, если при реальном физическом обмене происходит сбой. Т. е. необходимо, предположим, записать блок, он записывается в буфер, и получен ответ от системы, что обмен закончился успешно, но когда система реально запишет этот блок на ВЗУ, неизвестно. При этом может возникнуть нештатная ситуация, связанная с тем, что запись может не пройти, предположим, из-за дефектов носителя. Получается ситуация, при которой обращение к системе за функцией обмена для процесса прошло успешно (процесс получил ответ, что все записано), а, на самом деле, обмен не прошел.

Таким образом, эта система рассчитана на надежную аппаратуру и на корректные профессиональные условия эксплуатации. Для борьбы с вероятностью потери информации при появлении нештатных ситуаций, система достаточно «умна», и действует верно. А именно, в системе имеется некоторый параметр, который может оперативно меняться, который определяет периоды времени, через которые осуществляется сброс системных данных. Второе - имеется команда, которая может быть доступна пользователю, - команда SYNC. По этой команде осуществляется сброс данных на диск. И третье - система обладает некоторой избыточностью, позволяющей в случае потери информации, произвести набор действий, которые информацию восстановят или спорные блоки, которые не удалось идентифицировать по принадлежности к файлу, будут записаны в определенное место файловой системы. В этом месте их можно попытаться проанализировать и восстановить вручную, либо что-то потерять. Наш университет одним из первых в стране начал эксплуатировать операционную систему UNIX, и сейчас уже можно сказать, что проблем ненадежности системы, с точки зрения фатальной потери информации, не было.

8 Многомашинные ассоциации.

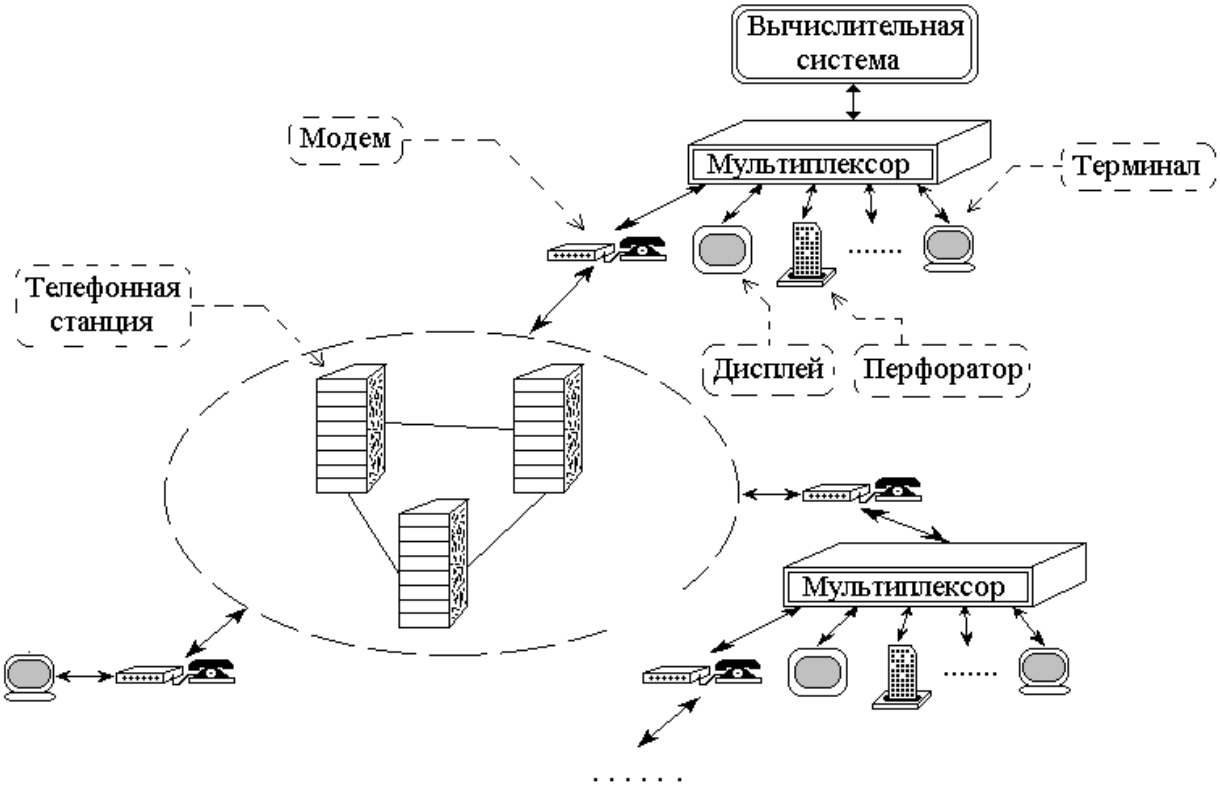
8.1 Общая характеристика, основные составляющие, примеры.

Одной из основных характеристик и свойств использования современный вычислительной техники является ее функционирование в составе многомашинных ассоциаций – некоторых программно-аппаратных комплексов, объединяющих в своем составе различное оборудование и программные системы. Состав и конкретные свойства многомашинных ассоциаций зависит от целей, которые ставятся перед данным образованием. Рассмотрим наиболее типичные примеры многомашинных комплексов.

Терминальные комплексы.

Исторически, одним из первых примеров многомашинных ассоциаций являлись терминальные комплексы. ***Терминальный комплекс*** это многомашинная ассоциация предназначенная для организации массового доступа удаленных и локальных пользователей к ресурсам некоторой вычислительной системы. При этом, к примеру, возможно использование терминальных комплексов для сбора и централизованной обработки информации (например, обработка результатов переписи населения или выборов) или для массового доступа удаленных пользователей к информации, размещенной в вычислительной системе (например, доступ пользователей к электронной библиотеке или система бронирования и продажи авиа или железнодорожных билетов). Временем появления подобных задач является конец 50-х – начало 60-х годов 20 века.

Структуру терминального комплекса можно примерно изобразить следующим образом.



Терминальный комплекс может включать в свой состав:

- ***основную вычислительную систему*** – систему, массовый доступ к ресурсам которой обеспечивается терминальным комплексом;
- ***локальные мультиплексоры*** – аппаратные комплексы, предназначенные для осуществление связи и взаимодействия вычислительной системы с несколькими устройствами через один канал ввода/вывода, в общем случае возможна схема М

x N, где M – число обслуживаемых мультиплексором устройств, N число используемых для организации работы каналов ввода/вывода (M > N);

- ***локальные терминалы*** – оконечные устройства, используемые для взаимодействия пользователей с вычислительной системой (это могут быть алфавитно-цифровые терминалы, графические терминалы, устройства печати, вычислительные машины, эмулирующие работу терминалов и т.п.) и, подключаемые к вычислительной системе непосредственно через каналы ввода/вывода или через локальные мультиплексоры;
- ***модемы*** – устройства, предназначенные для организации взаимодействия вычислительной системы с удаленными терминалами с использованием телефонной сети. В функцию модема входит преобразование информации из дискретного, цифрового представления, используемого в вычислительной технике в аналоговое представление, используемое в телефонии и обратно (в общем случае модем это устройство, предназначенное для взаимного преобразования данных из различных форм представления, например, могут быть оптические модемы, преобразующие данные из цифрового формата в оптический, предназначенный для передачи по оптоволоконным линиям связи). Со стороны вычислительной системы модем подключается либо через канал ввода/вывода, либо через мультиплексор.
- ***удаленные терминалы*** – терминалы, имеющие доступ к вычислительной системе с использованием телефонных линий связи и модемов.
- ***удаленные мультиплексоры*** – мультиплексоры, подключенные к вычислительной системе с использованием телефонных линий связи и модемов.

Телефонная сеть состоит из набора телефонных станций, объединенных друг с другом линиями связи. Связь абонентов телефонной в том числе и связь удаленных терминалов с вычислительной системой осуществляется с использованием ***коммутируемого канала,*** либо по ***выделенным каналам***. Суть соединения через коммутируемый канал заключается в том, что при нескольких звонках к одному и тому же абоненту, раз от раза маршруты коммутации (т.е. набор проводов, по которым идет сообщение) отличаются друг от друга, за счет того, что каждый раз выбираются свободные каналы в телефонных станциях по пути соединения. После завершения сеанса связи между абонентами коммутируемый канал освобождается. При использовании выделенного канала маршрут коммутации между абонентами фиксируется на период аренды выделенного канала. Достоинства/недостатки использования коммутируемых и выделенных каналов очевидны.

Линия связи, которая связывает один удаленный терминал с компьютером, называется линией связи типа ***точка-точка***. Таким образом эта линия может быть либо выделенной (мы договариваемся с телефонными станциями и фиксируем коммутацию), либо коммутируемой.

Канал может быть ***многоточечным***. При этом на входе находится удаленный мультиплексор. Многоточечные каналы также могут быть либо выделенными, либо коммутируемыми.

С точки зрения организации потоков информации можно выделить следующие разновидности каналов.

1. ***Симплексные каналы*** - каналы, по которым передача информации ведется в одном направлении (например, телевизионный канал – обеспечивает передачу информации только в одном направлении от передающей антенны к принимающей).
2. ***Дуплексные каналы*** - каналы, которые обеспечивают одновременную передачу информации в двух направлениях (например, телефонный разговор, мы одновременно можем и говорить и слушать).

3. ***Полудуплексные каналы*** - каналы, которые обеспечивают передачу информации в двух направлениях, но в каждый момент времени только в одну сторону (подобно рации).

Одним из примеров терминального комплекса может быть система NASDAQ (National Association of Securities Dealers Automated Quatation), построенная в 60-70-х годах 20 века и предназначенная для сбора и передачи сообщений о курсах акций на бирже. Система была построена на использовании мощной, по тем временам, вычислительной машины Univac-1108 и значительного числа терминалов (несколько тысяч), установленных в биржевых конторах по всей территории США.

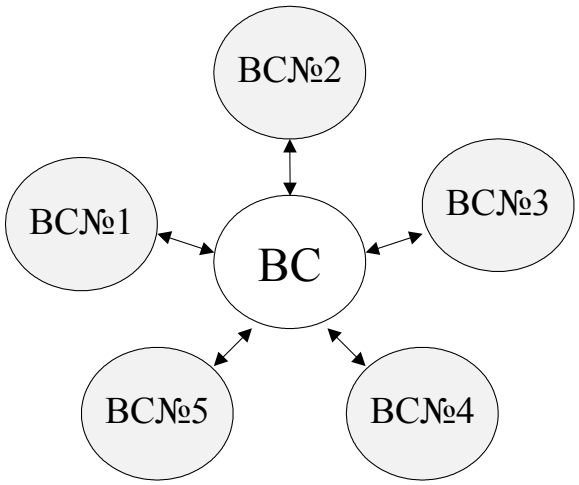
Многомашинные вычислительные комплексы

Многомашинные вычислительные комплексы (ММВК) - это программно аппаратное объединение группы вычислительных машин, в которых:

1. На каждой из машин работает своя операционная система (этот признак отличает ММВК от многопроцессорного вычислительного комплекса).
2. В ММВК имеются общие физические ресурсы, например ОЗУ, ВЗУ или общие каналы связи (а, следовательно, имеются проблемы синхронизации доступа).

ММВК использовались в качестве систем сбора и обработки больших наборов данных, и для организации глобальных терминальных комплексов. ММВК появились в начале 60-х и сейчас продолжают успешно существовать. Одно из основных применений ММВК - это дублирование вычислительной мощи, примером таких систем может служить любая система управления важными технологическими процессами.

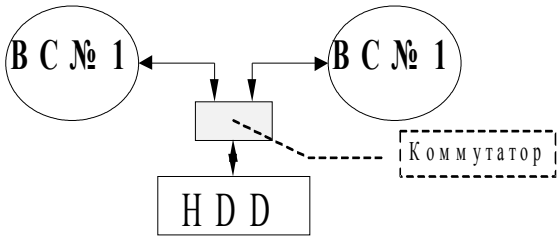
ММВК может выглядеть и так:



В ММВК общий ресурс является общим не только для всех ВС, но и для групп ВС, благодаря этому мы можем организовывать ММВК сложной структуры, необходимой для решения конкретной проблемы (Например, ММВК для продажи авиабилетов и ММВК для параллельного проведения какого-нибудь сложного научного расчета). Мы также обсудили тот факт, что в ММВК на каждой из машин работает своя операционная система. Отсюда вытекает, что все проблемы взаимодействия должны решаться на уровне взаимодействия ОС. Система, аналогичная ММВК, но в которой работает одна ОС, - многопроцессорная ВС. Существуют задачи, для которых не хватает средств, предоставляемых терминальными

комплексами. Это, например, проблема организации больших баз данных. В этом случае используют ММВК.

В ММВК имеется проблема синхронизации доступа к разделяемым ресурсом. Разделяемыми ресурсами могут быть устройства внешней памяти, ОЗУ, каналы связи, соединенные двумя или более компонентами вычислительного комплекса. Рассмотрим такой пример. У нас есть ММВК, состоящий из двух ВС. Разделяемый ресурс - жесткий диск. Проблема в данном случае явно формулируется так: «Нужно научить две ВС синхронизированно обмениваться с HDD.» Т. е. если программа одной ВС что-то пишет на HDD, то область данных, в которую она пишет или весь HDD должны быть заблокированы для другой ВС (Проблема напоминает проблему семафоров). Одно из решений - коммутатор HDD, некий контроллер, который имеет команду, блокирующую HDD. При начале обмена одной вычислительной системы доступ к HDD заблокирован для других ВС. А эта ВС в монопольном режиме использует HDD. Если другая ВС попытается начать обмен с HDD возможны два решения:



1) синхронное ожидание;
2) асинхронное ожидание (система не будет простаивать, она временно остановит процесс, подавший заказ на обмен и активизирует другой процесс).

На самом деле коммутаторы, конечно, более интеллектуальны. Они, например, устанавливают блокировку не на весь HDD, а только на некоторые его блоки.

Приведенное выше решение привлекает своей технической простотой как с аппаратной точки зрения, так и с точки зрения программной реализации (нет сложных взаимосвязей), но оно имеет существенный недостаток. ВС может заблокировать HDD и после этого зациклиться. Для борьбы с такими ситуациями можно использовать различные устройства, отличные от коммутатора HDD, позволяющие послать сигнал от одной ВС к другой. Это может быть, например, низкоскоростной канал связи (скорость передачи нам здесь не нужна).

8.2 Сети ЭВМ. Организация взаимодействия в сети. Модель ISO/OSI.

Вычислительная сеть или ***сеть ЭВМ*** есть, в некотором смысле, развитие и обобщение терминальных комплексов и многомашинных вычислительных комплексов. Вычислительная сеть представляет собой программно-аппаратный комплекс обладающий следующими основными характеристиками:

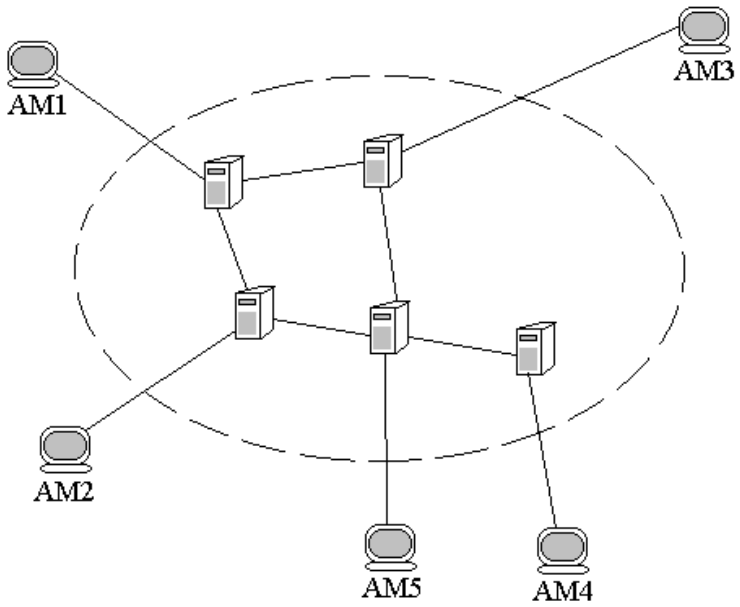
- сеть может состоять из значительного числа взаимодействующих друг с другом ЭВМ, обеспечивающих сбор, хранение, обработку и передачу информации;
- сеть ЭВМ предполагает возможность распределенной обработки информации;
- расширяемость сети – возможность развития сети по протяженности (размещению), по расширению пропускной способности каналов связи, по количеству и производительности ЭВМ;
- возможность применения симметричных интерфейсов обмена информацией между ЭВМ сети.

В общем случае, будем считать, что сеть состоит из двух разновидностей ЭВМ:

- ***абонентские или основные ЭВМ***, которые обеспечивают информационно-вычислительные услуги сети;
- ***коммуникационные или вспомогательные ЭВМ***, обеспечивают выполнение всех служебных функций по преобразованию и передаче информации.

В реальности, в современных сетях ЭВМ функции абонентских и коммуникационных ЭВМ совмещаются.

Для обобщения определения сети ЭВМ рассмотрим пример. Пусть дана сеть, состоящая из абонентских и коммуникационных ЭВМ. Абонентские машины могут осуществлять взаимодействие друг с другом через ***коммуникационную среду или коммуникационную сеть***. Коммуникационная среда включает в себя ***каналы передачи данных***, обеспечивающие взаимодействие между машинами и коммуникационными машинами. Конкретная топология сети зависит от назначения данной сети и определяется составом абонентских ЭВМ и топологией коммуникационной среды. Итак, абонентские машины могут осуществлять взаимодействие друг с другом через коммуникационную среду, в рамках которой используются каналы передачи данных и коммуникационные машины.



Существует классическое разделение сетей на три типа: сеть коммутации каналов, сеть коммутации сообщений и сеть коммутации пакетов. Рассмотрим основные свойства каждой из перечисленных разновидностей сетей.

Сеть коммутации каналов. Сеть строится на основе использования коммутируемых каналов (см. Терминальные комплексы), т.е. для обеспечения сеанса связи двух абонентских ЭВМ на время всего сеанса выделяется коммутируемый канал – устанавливается прямое соединение между взаимодействующими абонентскими ЭВМ. Для этих целей осуществляется поиск пути в сети, по которому будет происходить соединение (при наличии нескольких путей выбирается какой-то один из них; на том, по какому именно критерию выбирается путь в случае альтернативы, мы останавливаться не будем). На время сеанса связи найденный путь считается монопольно выделенным для этих двух машин. Достоинства такого вида сети - простота реализации и эффективность работы в случае успешной коммутации, так как скорость взаимодействия между машинами равна скорости самого медленного компонента сети, участвующего в связи (это максимально возможная скорость). Главный недостаток заключается в том, что такая связь может блокировать другие соединения. Уйти от этой проблемы можно потребовав от коммутационной среды большой избыточности, т.е. организовать дополнительные (дублирующие) каналы.

Сеть коммутации сообщений. Если коммутация каналов - это коммутация на время всего сеанса связи, то коммутация сообщений - это связь, при которой весь сеанс разделяется на передачу сообщений (сообщение - некоторая, логически завершенная, порция данных). Взаимодействие ЭВМ в данных сетях осуществляется в терминах передачи сообщений. При этом для передачи сообщения не требуется установления прямого соединения между взаимодействующими абонентскими ЭВМ. Сообщение начинает передаваться по сети по мере освобождения каналов в коммуникационной среде (***абонентская_ЭВМ⇔коммуникационная_ЭВМ***; ***коммуникационная_ЭВМ⇔коммуникационная_ЭВМ***). При этом на коммуникационные машины ложится значительная дополнительная нагрузка – коммуникационные ЭВМ должны обеспечивать буферизацию сообщений, так как прямого

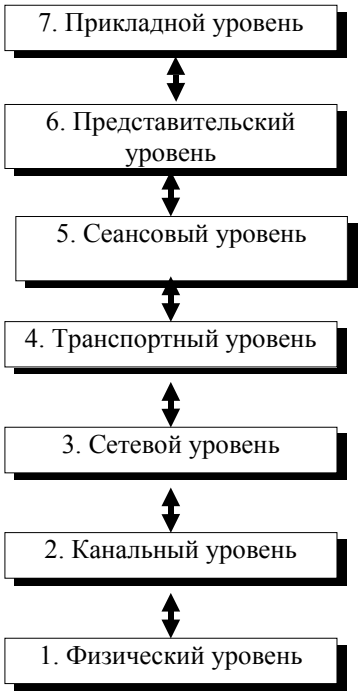
соединения не устанавливается и возможна ситуация при которой канал, необходимый для продолжения передачи сообщений занят. Достоинства - логическая и физическая простота, снижение монопольного фактора по сравнению с сетью коммутации каналов. Недостатки - снижение скорости работы в сети, отсутствие детерминированности в определении времени доставки информации, необходимость существенного увеличения мощности коммуникационных ЭВМ для обеспечения буферизации.

Сеть коммутации пакетов. Сеанс разбивается на сообщения, сообщения, в свою очередь, разбиваются на порции данных одинакового объема - пакеты. По сети перемещаются не сообщения, а пакеты. Здесь действует принцип горячей картошки: основное действие коммутационной машины - как можно быстрее избавиться от пакета, определив кому его далее можно «перекинуть». Поскольку все пакеты одинакового объема, то не возникает проблем с буферизацией, потому что мы всегда можем рассчитать необходимую буферную способность коммутационных машин. Логически происходит достаточно быстрое соединение, потому что сеть коммутации пакетов практически не имеет ситуаций, когда какие-то каналы заблокированы на продолжительное время. За счет того, что происходит дробление сеанса на пакеты, имеется возможность оптимизации обработки ошибок при передаче данных. Если возникает ошибка в режиме коммутации каналов, то надо повторить весь сеанс, если в режиме коммутации сообщений, то надо повторить сообщение, при коммутации пакетов достаточно повторить передачу пакета, в котором обнаружена ошибка.

В реальных системах используются многоуровневые сети, которые в каких-то режимах работают в режиме коммутации каналов, в каких-то режимах работают в режиме коммутации сообщений и т.д.. На сегодняшний день можно сказать, что сетей, однозначно принадлежащих к одному из вышеперечисленных типов, нет, а используется их комбинация.

Важным понятием, используемым при функционировании сетей ЭВМ является понятие протокола связи. В общем случае существует множество определений понятия ***протокол***. Рассмотрим одно из простейших. ***Протокол*** - формальное описание сообщений и правил, по которым сетевые устройства (вычислительные системы) осуществляют обмен информацией.

Развитие многомашинных ассоциаций вообще, и сетей ЭВМ в частности, определило возникновение необходимости стандартизации взаимодействия, происходящего в сети. Развитие любого технического новшества всегда испытывает трудности в связи с отсутствием единого стандарта. Поэтому в конце 70-х начале 80-х годов ISO (International Standard Organization) предложила т.н. стандарт взаимодействия открытых систем ISO/OSI (Open System Interface), который должен был стандартизировать основные интерфейсы, позволяющие строить и развивать сети ЭВМ. Была предложена семиуровневая модель организации взаимодействия в сети.



Каждый уровень представляет собой условный, логически целостный набор действий и форматов данных, предназначенных для передачи данных взаимодействующих в сети вычислительных систем. В некоторых случаях предполагалась регламентация взаимодействия на уровне аппаратных компонентов вычислительных систем, в некоторых – программных.

В общем случае, каждый из уровней модели ISO/OSI есть абстракция, которой может быть поставлено в соответствие некоторое количество протоколов данного уровня. Т.е. каждый протокол, реализованный в соответствии с данной моделью принадлежит некоторому единственному уровню, но, вместе с тем, каждому уровню модели может соответствовать произвольной количество протоколов.

Рассмотрим общие характеристики и назначение каждого из уровней модели ISO/OSI.

Физический уровень

На этом уровне решаются вопросы взаимосвязи в терминах сигналов. Этот уровень однозначно определяется физической средой, используемой для передачи данных и отвечает за организацию физической связи между устройствами и передачи данных в сети.

Канальный уровень

Этот уровень по-прежнему сильно ориентирован на конкретную физическую среду. Он управляет доступом к физической среде передачи данных, осуществляет синхронизацию передачи. Здесь формализуются правила передачи данных, решаются задачи обнаружения и локализации ошибок.

Сетевой уровень

Этот уровень управляет связью в сети между двумя взаимодействующими машинами. Здесь также решаются вопросы, связанные с маршрутизацией и адресацией в сети.

Транспортный уровень

Данный уровень иногда называют уровнем логического канала. На этом уровне решаются проблемы управления передачей данных и связанные с этими проблемами задачи – локализация и обработка ошибок, сервис передачи данных.

Сеансовый уровень

Это уровень управления сеансами связи между взаимодействующими программами. На этом уровне решаются проблемы синхронизации отправки и приема данных,

прерывания/продолжения работы в тех или иных внештатных ситуациях, управление подтверждением полномочий (паролей).

Представительский уровень

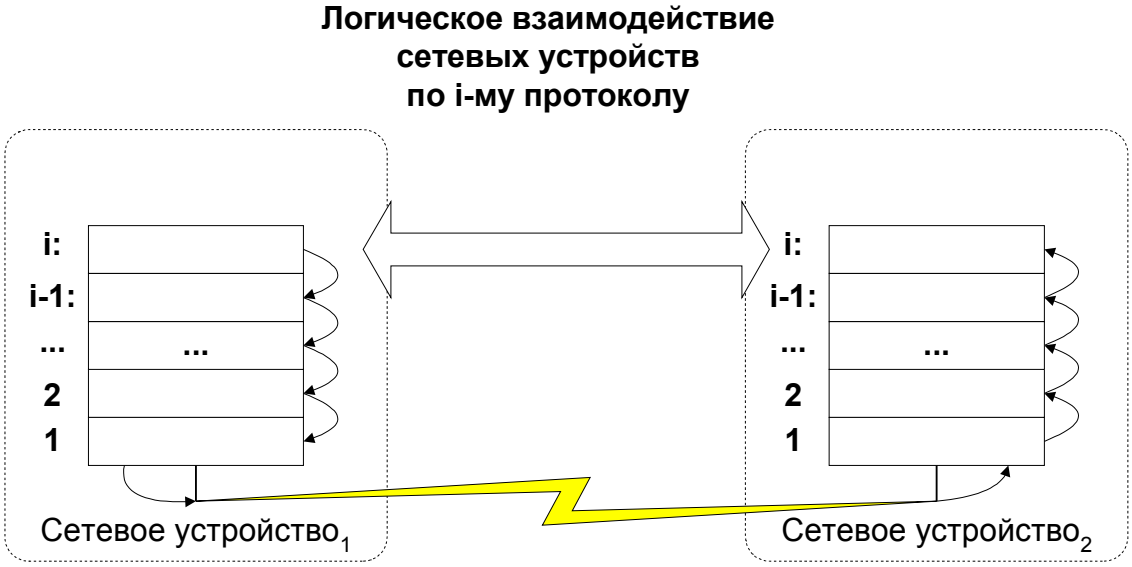
Уровень представления данных. На этом уровне находятся протоколы, реализующие единые соглашения перевода из внутреннего представления данных конкретной машины в сетевое и обратно.

Прикладной уровень

Этот уровень осуществляет стандартизацию взаимодействия с прикладными системами.

Каждый протокол одной вычислительной системы общается с одноименным протоколом на другой вычислительной машине (следует отметить, что здесь мы несколько упрощаем ситуацию, так как не всегда в общении находятся две ВС, в реальности общение через протокол может осуществляться и в рамках одной ВС и трех и более ВС).

Заметим, что несмотря, на декларацию о том, что общение ВС в сети происходит между одноименными протоколами, реальная обработка взаимодействия заключается в последовательной передаче сообщения от протокола i-го уровня на одной ВС к i-1, i-2,… и, наконец, физическому уровню на этой же ВС, который должен обеспечить физическую передачу сообщения физическому уровню ВС – получателя, и последовательному «подъему» по уровням ВС – получателя, до соответствующего одноименного протокола i-го уровня.



Для организации взаимодействия при передаче сообщений от одного уровня к соседнему, существуют стандартизованные соглашения, которые называются ***интерфейсами***.

Таким образом, данные от одной прикладной программы до другой прикладной программы в сети проходят путь от уровня протоколов прикладных программ до физического уровня на ВС, отправляющей данные, и далее на ВС, принимающей данные, они проходят этот путь обратном порядке.

Последовательность протоколов от максимально реализованного уровня до физического образуют ***стек протоколов,*** реализованный на данной ВС. В стеке протоколов предполагается реализация всех протоколов от максимального до физического уровней и их взаимосвязь через соответствующие интерфейсы. Т.е. не допускается ситуация наличия, например, протоколов 5, 4, 2, и 1 при отсутствии протокола 3 уровня. Стек протоколов не обязан содержать протоколы всех семи уровней. Т.е. возможна, ситуация, при которой в ВС

реализованы только протоколы до 4 (или любого другого) уровня, это означает, что данная система, а точнее протоколы, входящие в данный стек, могут общаться со стеком, содержащим не менее 4 уровней. Уточним понятие взаимодействия в сети. Взаимодействие организуется между стеками протоколов и их реализации могут размещаться, как в пределах одной ВС, так и на различных ВС.

Предложенная модель организации взаимодействия в сети, основанная на стандартизации взаимодействия в пределах одноименных уровней и стандартизации передачи данных через интерфейсы, позволила создать основу для организации открытых к развитию и модернизации сетей ЭВМ. Реальных систем, построенных в полном объеме по модели ISO/OSI нет, так как, в итоге, данная модель являлась рекомендацией и не содержала декларации всех своих протоколов и интерфейсов. Однако, появление модели ISO/OSI дало практический толчок к бурному развитию стандартов протоколов и интерфейсов открытых систем.

8.3 Семейство протоколов TCP/IP.

Рассмотрим что представляет из себя семейство протоколов TCP/IP(Transfer Control Protocol/Internet Protocol). Оно обладает следующими свойствами:

- открытые (доступные для использования) стандарты протоколов, широко поддерживаемые разными вычислительными платформами и операционными системами;
- независимость от аппаратного обеспечения сети передачи данных, TCP/IP может работать и объединять вместе сети, построенные на Ethernet, X.25, телефонных линиях связи и вообще на любых типах носителей, передающих данные;
- общая схема именования сетевых устройств, которая позволяет любому устройству единственным образом адресовать любое другое устройство в сети Internet;
- стандартизованные протоколы прикладных программ.

Рассмотрим основные протоколы TCP/IP, сравнивая их с протоколами модели ISO/OSI.

8.3.1 Архитектура семейства TCP/IP

Протоколы семейства TCP/IP не следуют строго модели ISO/OSI. Они разбиты на *четыре* уровня.

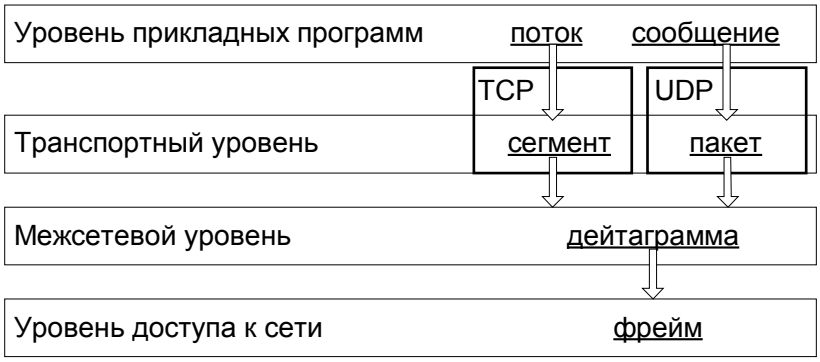
Уровень модели TCP/IP	Уровень модели OSI
4. Уровень прикладных программ Состоит из прикладных программ и процессов, использующих сеть и доступных пользователю. В отличие от модели OSI, прикладные программы сами стандартизуют представление данных.	**Уровень прикладных программ** **Уровень представления данных**
3. Транспортный уровень Обеспечивает доставку данных от компьютера к компьютеру. Кроме того, на этом уровне существуют средства для поддержки логических соединений между прикладными программами. В отличие от транспортного уровня модели OSI, в функции транспортного уровня TCP/IP не всегда входят контроль за ошибками и их коррекция. TCP/IP предоставляет два разных сервиса передачи данных на этом уровне. Протокол TCP обеспечивает все вышеперечисленные функции, а UDP – только передачу данных.	**Сеансовый уровень** **Транспортный уровень**
2. Межсетевой уровень Работает с дейтаграммами, адресами, выполняет маршрутизацию и «прикрывает» транспортный уровень от общения с физической сетью. Однако, в отличие от сетевого уровня модели OSI, этот уровень не устанавливает соединений с другими машинами.	**Сетевой уровень**
1. Уровень доступа к сети Состоит из подпрограмм доступа к физической сети. Модель TCP/IP не разделяет два уровня модели OSI – канальный и физический, а рассматривает их как единое целое.	**Канальный уровень** **Физический уровень**

Так же, как и в модели ISO/OSI, в TCP/IP данные проходят путь от уровня прикладных программ к уровню доступа к сети при передаче данных и в обратном порядке при их получении. Каждый уровень TCP/IP добавляет к исходной порции данных свою контрольную информацию для обеспечения правильной доставки. Эта контрольная информация называется заголовком, потому что помещается перед посылаемыми данными. При передаче каждый следующий уровень рассматривает порцию данных, пришедшую от предыдущего, как единое целое и помещает перед ней свой заголовок.

После того, как данные переданы по сети и получены уровнем доступа к сети, происходит обратный процесс. Каждый уровень вырезает свой заголовок из порции данных, а после этого передает оставшуюся часть следующему уровню. Таким образом, при прохождении данных снизу вверх (то есть при ее расшифровке), они рассматривается уже не как единое целое, а как совокупность заголовка и некоторой информации.

Протоколы каждого из уровней оперируют порциями данных, имеющих зависящие от конкретного уровня названия и структуру.
Протоколы уровня доступа к сети используют при передаче и приеме данных пакеты, называемые ***фреймами.*** На межсетевом уровне используются ***дейтаграммы***. Уровень транспортных протоколов семейства представляется двумя протоколами TCP и UDP.

Протокол TCP оперирует ***сегментами***. UDP – ***пакетами***. На уровне прикладных программ, системы построенные на использовании протокола TCP используют ***поток*** данных, а системы использующие UDP - ***сообщения***.



Рассмотрим, кратко, функции каждого из уровней протокола TCP/IP.

8.3.2 Уровень доступа к сети

Уровень доступа к сети является самым нижним уровнем в иерархии протокола TCP/IP. Протоколы на этом уровне обеспечивают систему средствами для передачи данных другим устройствам в сети. Они определяют, как использовать сеть для передачи дейтаграмм IP. В отличие от протоколов более высоких уровней, протоколы этого уровня должны знать детали физической сети (структуру пакетов, систему адресации и т.д.), чтобы правильно оформить передаваемые данные.

TCP/IP разработана таким образом, чтобы протоколы более высоких уровней не зависели от протоколов нижних уровней. Для уровня доступа к сети протоколы IP, TCP, UDP и т.д. являются протоколами более высокого уровня. Благодаря этому при появлении новых сетевых аппаратных средств необходима только разработка новых протоколов доступа к сети. Поэтому существует множество протоколов этого уровня – по одному на каждый стандарт сети (например, протокол передачи пакетов IP по сети Ethernet или протокол передачи этих же пакетов через последовательный порт, через модемное соединение).

8.3.3 Межсетевой уровень

8.3.3.1 Протокол IP.

Протокол IP самым важным протоколом межсетевого уровня. Функции этого протокола включают в себя:

- формирование *дейтаграмм*;
- поддержание системы *адресации*;
- обмен данными между транспортным уровнем и уровнем доступа к сети
- организацию *маршрутизацию* дейтаграмм
- *разбиение и обратную сборку* дейтаграмм

Прежде чем приступить к детальному описанию каждой функции протокола, рассмотрим некоторые его характеристики.

IP является протоколом ***без логического установления соединения***. Это значит, что он не обменивается контрольной информацией для установки соединения, перед началом передачи данных. IP оставляет другим протоколам право устанавливать

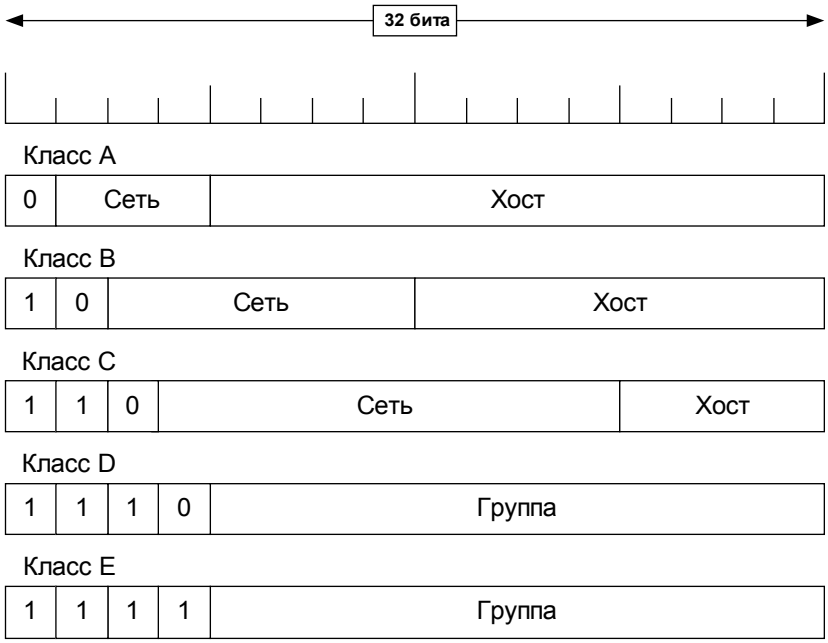
соединения – этим занимается либо протокол TCP, либо сами прикладные программы.

Протокол IP *не занимается обнаружением и исправлением ошибок*. Проверку полученных данных должны осуществлять другие протоколы, если это вообще необходимо.

Одним из основных свойств протокола IP является система адресации, которая обеспечивает уникальное именование любого ***сетевого устройства*** (устройство будем считать сетевым, если с ним ассоциирован некоторый стек протоколов). Рассмотрим структуру IP адреса.

IP адрес представляется последовательностью четырех байтов. В адресе кодируется уникальный номер сети, а также номер компьютера (сетевого устройства в сети). Следует отметить, что существуют специальные IP адреса, информационные поля которых интерпретируются по другому, но мы не будем акцентировать на них свое внимание. Для представление содержимого IP адреса используется последовательность цифр:

$N_1.N_2.N_3.N_4$,
где N_i – десятичное представление содержимого i – го байта адреса.



1. ***IP адрес класса A.*** Первый бит адреса IP равен 0. Следующие 7 бит определяют номер сети, а остальные 24 – номер компьютера в этой сети ($1 \le N_1 \le 126$).
2. ***IP адрес класса B.*** Первые два бита адреса IP равны 10. Следующие 14 бит определяют адрес сети, остальные 16 – адрес хоста ($128 \le N_1 \le 191$).
3. ***IP адрес класса C.*** Первые три бита адреса IP - 110. Следующие 21 бит определяют адрес сети, оставшиеся 8 – адрес хоста ($192 \le N_1 \le 223$).
4. ***IP адрес класса D.*** Первые четыре бита адреса IP – 1110 . Используется для служебных целей (групповая передача данных).
5. ***IP адрес класса E.*** Первые четыре бита адреса IP – 1111 . Используется для служебных целей.

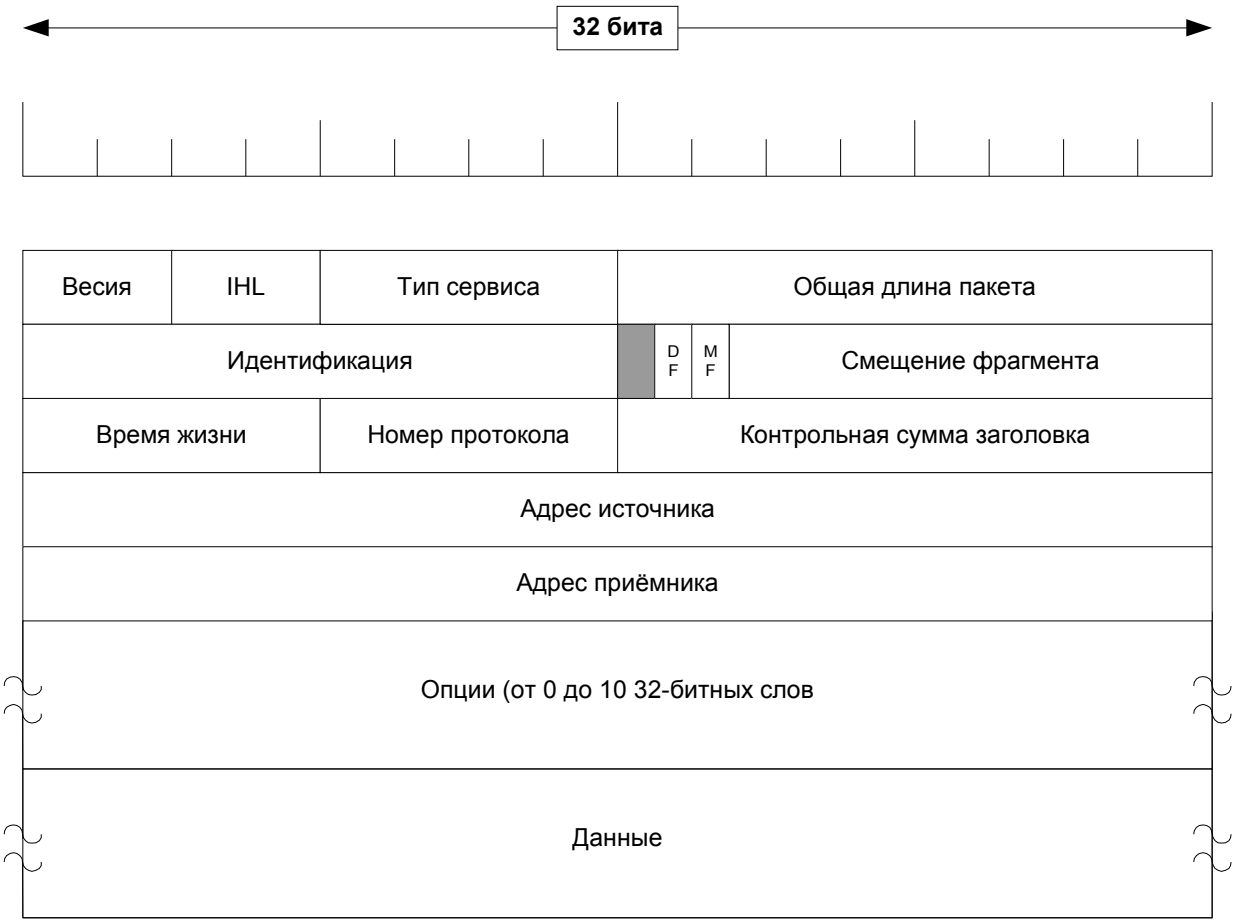
Некоторые из IP адресов являются зарезервированными, т.е. их интерпретация отличается от стандартной.

Поле номера сети	***Поле номера машины/устройства***	***Комментарий***
Все нули	Все нули	Адрес данного устройства
Номер сети	Все нули	Ссылка на сеть в целом.
Все нули	Номер устройства	Устройство в данной сети
Все единицы	Все единицы	Все устройства данной сети
Номер сети	Все единицы	Все устройства заданной сети
127_{10}	Код	Используется для отладки и тестирования сетевых приложений (*зацикленный адрес* - loopback address). При отправке данных по этому адресу, стек протоколов возвращает переданные данные процессу-отправителю. Т.е. происходит эмуляция работы сети, без реального сетевого взаимодействия (взаимодействия между различными стеками протоколов).

Часто IP адреса называют адресами ***хостов*** или компьютеров. Но на самом деле, это не совсем точно. IP-адреса соответствуют сетевым интерфейсам компьютера, а не самому компьютеру. ***Шлюзы*** могут иметь несколько IP-адресов одновременно – по одному на каждый интерфейс, через который они общаются с какой-либо из подключенных к этому шлюзу сетей. Как используются адреса? IP использует адрес сети (первую часть IP-адреса) для маршрутизации дейтаграмм между сетями. Полный адрес, включая адрес хоста, используется для окончательной доставки дейтаграммы до компьютера, когда она уже достигла нужной сети.

Дейтаграммы

Протоколы TCP/IP были созданы для передачи данных через ARPANET, которая является сетью с коммутацией пакетов. Пакет – это блок данных, который передаётся вместе с информацией, необходимой для его корректной доставки. Каждый пакет перемещается по сети независимо от остальных.



Дейтаграмма – это пакет протокола IP. Контрольная информация занимает первые пять или шесть 32-битных слов дейтаграммы. Это её заголовок (header). По умолчанию, его длина равна пяти словам, шестое является дополнительным. Для указания точной длины заголовка в нём есть специальное поле – *длина заголовка* (IHL, Internal Header Length).

В поле *версия* записана версия протокола IP. Это поле нужно для того, чтобы можно было отличать друг от друга дейтаграммы разных версий протокола IP.

Поле *тип сервиса* теоретически предоставляет хосту возможность выбирать, какой тип сервиса он хочет получить от протоколов сетевого уровня. В этом поле можно указать различные комбинации надёжности и скорости доставки.

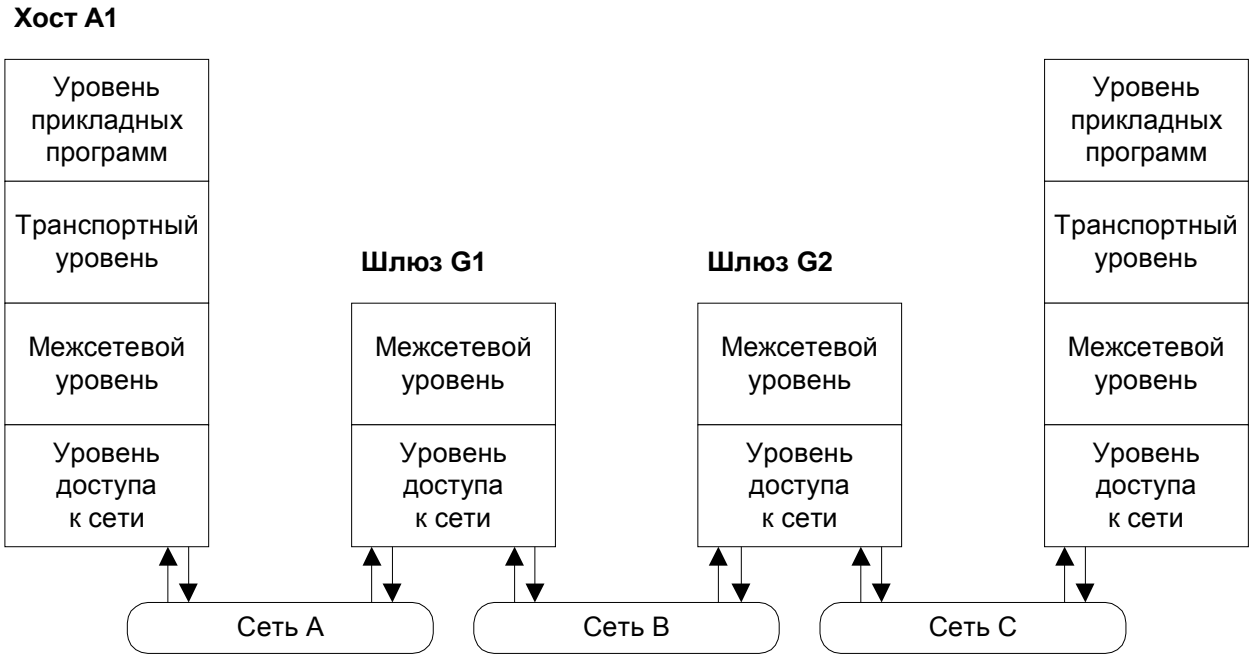
IP доставляет дейтаграммы по адресу, записанному в поле *адрес назначения* (Destination Address) в слове 5 заголовка. Адрес назначения – это стандартный 32-битный адрес протокола IP. Он определяет номер сети назначения и номер хоста в этой сети. Если хост находится в локальной сети, то пакет доставляется сразу по месту назначения. Если нет, то пакет сначала отправляется на межсетевой шлюз (маршрутизатор). Шлюз – это устройство, передающее пакеты между различными сетями. Процесс выбора шлюза или маршрутизатора называется *маршрутизацией*. Протокол IP отвечает за маршрутизацию каждого пакета.

Маршрутизация дейтаграмм

TCP/IP оперирует только двумя типами сетевых устройств: шлюзами (или маршрутизаторами) и хостами. Шлюзы перемещают данные между различными сетями, а хосты нет. Но если хост подключен больше чем к одной сети (это так называемый multihomed хост), он может передавать пакеты как в одну сеть, так и в другую. Однако он не будет делать этого для других компьютеров в сети, т.е. не будет заниматься маршрутизацией пакетов. В этом его отличие от шлюза.

Компьютерные системы могут передавать данные только внутри той сети, к которой они подключены. Поэтому передача дейтаграмм из одной сети в другую идёт через шлюзы – от одного к другому. Внутри хоста данные проходят пути от уровня прикладных программ до уровня доступа к сети (или наоборот). Дейтаграммы, которые переправляет шлюз, поднимаются только до межсетевого уровня. На этом уровне протокол IP, узнавая адрес получателя данных (на протяжении всего пути следования этот адрес не меняется – меняются промежуточные машины), принимает решение отправить дейтаграмму в одну из сетей, к которым подключен.

На рисунке показано, как используются шлюзы для ретрансляции пакетов. Внутри хостов (будем называть их конечными системами) данные проходят все четыре уровня, а внутри шлюзов (промежуточных систем) они поднимаются только до межсетевого уровня, где протокол IP и принимает решение о маршрутизации.



Поскольку данные (дейтаграммы) могут передаваться непосредственно только между устройствами, объединёнными в одну физическую сеть, то в нашем примере передача пакетов от хоста A1 хосту C1 осуществляется в несколько приёмов, через шлюзы G1 и G2. Сначала хост A1 посылает данные шлюз G1, с которым объединён в сетью A. Шлюз G1 по сети B пересылает данные шлюзу G2. Шлюз G2, наконец, пересылает данные хосту C1, благодаря тому, что они оба подключены к сети C. Очень важно, что хост A1 ничего не знает о шлюзах, находящихся дальше шлюза G1 (то есть о шлюзах, с которыми он не объединён в одну сеть). Он просто посылает данные, адресованные C1 (или вообще любому хосту в сети B или C), этому шлюзу и полностью полагается на его способность правильно ретранслировать пакеты в соответствующую сеть. Точно так же хост C1 не имеет ни малейшего понятия о шлюзе G1 и полностью полагается на свой локальный шлюз G2 при пересылке данных хостам в сети A или B.

С помощью поля *время жизни* (TTL, Time to live) можно указать, как долго дейтаграмма может прожить в сети. При прохождении через маршрутизаторы это поле уменьшается, как правило на 1 (если дейтаграмма задержалась в маршрутизаторе больше, чем на 1 секунду, значение поля уменьшается больше чем на 1). Как только *время жизни* уменьшится до 0, маршрутизатор обязан уничтожить дейтаграмму и не передавать её дальше.

Фрагментация дейтаграмм

Когда Internet Protocol готовит данные, полученные от транспортного уровня, он может обнаружить, что сегмент (или пакет) данных не может быть передан по сети целиком. Тогда он принимает решение разбить его на несколько частей – фрагментов. Этот процесс, естественно, называется *фрагментацией*.

Аналогичная проблема может возникнуть и у шлюза. Дело в том, что каждая физическая сеть имеет так называемый *максимальный передаваемый блок* (MTU, maximum transmission unit), больше которого она ничего передать не может. Разные сети имеют разные MTU, поэтому шлюз может принять решение о фрагментации и в этом случае. Когда промежуточная машина либо хост назначения получают несколько дейтаграмм, являющимися фрагментами одного и того же сегмента (пакета) данных, происходит обратный процесс – *сборка.* Транспортный уровень этой машины (или хоста) получает свою порцию данных в нетронутом виде – так, как её передал исходный хост.

Передача дейтаграмм транспортному уровню

Когда протокол IP какого-нибудь хоста получает дейтаграмму, он должен передать данные, содержащиеся в ней, соответствующему протоколу транспортного уровня. Для определения протокола используется поле *номер протокола*(Protocol Number) из третьего слова заголовка дейтаграммы. Каждый протокол транспортного уровня имеет свой уникальный номер, по которому его и отыскивает протокол IP.

8.3.3.2 Протокол ICMP

Неотъемлемой частью IP и межсетевого уровня является Протокол контрольных сообщений Internet (ICMP, Internet Control Message Protocol). Он пользуется сервисом передачи дейтаграмм протокола IP для приёма и посылки собственных сообщений. Эти сообщения выполняют следующие функции:

Контроль за трафиком

Иногда возникает ситуация, когда количество получаемых дейтаграмм становится слишком большим, так что хост или шлюз не успевают их обработать. Тогда отправителю дейтаграмм посылается сообщение Source Quench Message, которое просит временно приостановить пересылку.

Обнаружение недостижимых получателей

Если система обнаруживает, что по какой-либо причине не может отправить дейтаграммы получателю, она оповещает отправителя. В этом случае передаётся *сообщение о недоступности получателя* (Destination Unreachable Message). Если недостижимый получатель – хост, то сообщение посылает промежуточный шлюз; если протокол транспортного уровня или прикладная программа – сам хост-получатель.

Изменение маршрута дейтаграмм

Шлюз может послать *сообщение о переназначении* (Redirect Message), чтобы попросить отправителя дейтаграмм использовать другой шлюз (возможно, из-за того, что он быстрее). При этом оба шлюза и отправитель должны быть соединены напрямую. Это связано с идеологией маршрутизации в Internet.

Проверка связи с хостом

С помощью ICMP можно узнать, есть ли связь с каким-нибудь устройством – шлюзом или хостом. Для этого достаточно послать сообщение Echo Message. Дело в том, что если хост или шлюз получают такое сообщение, они обязаны послать его обратно отправителю. Поэтому если ответ приходит – значит устройство работает и связь с ним есть. Если нет

ответа – то с этим устройством лучше пока не связываться – всё равно ничего не получится. Кстати, команда ping ОС UNIX работает используя именно это сообщение.

По сути, протокол ICMP является надстройкой над IP, которая выполняет информационные функции, контроль за работой, и диагностирование ошибок.

8.3.3.3 Протоколы разрешения адресов на примере адресации в Ethernet (ARP, RARP)

В самом начале этой главы вскользь было упомянуто о том, что IP-адрес имеет сетевой интерфейс, посредством которого устройство обменивается IP-дейтаграммами с другими компьютерами. Однако об этом адресе знает межсетевой уровень, но не имеет представления сетевой уровень. Это происходит из-за того, что в каждой физической сети используется своя схема адресации, отличная от адресации в Internet. Причём сколько разных типов физических сред – столько типов адресации. Сразу возникает вопрос – как адресовать данные, если известен только IP-адрес их получателя? Для этого нужно решить задачу отождествления физических адресов сетевых устройств и их IP-адресов.

Мы рассмотрим один из способов решения этой задачи на примере Протокола разрешения адресов (ARP, Address Resolution Protocol). Этот протокол транслирует IP-адреса в адреса Ethernet. Эти адреса состоят из 6 байт и гарантируют уникальность, то есть в мире нет двух Ethernet карт с одинаковыми адресами. Для трансляции IP-адреса в Ethernet-адрес ARP внутри себя содержит таблице соответствия одних адресов другим. Эта таблица строится автоматически. Когда ARP получает запрос на трансляцию, он сначала просматривает свою таблицу. Если IP-адрес в ней уже есть, то возвращается соответствующий ему адрес Ethernet. Если адреса в таблице ещё нет, то ARP выполняет следующие действия:

1. всем компьютерам сети Ethernet рассылается широковещательный запрос (в сетях Ethernet это возможно) с IP-адресом машины, Ethernet-адрес которой нужно узнать.
2. машины получают запрос. Если одна из них в IP-адресе узнаёт свой собственный адрес, она отсылает обратно (по Ethernet-адресу спрашивающей машины) пакет со своим Ethernet-адресом.
3. ARP получает пакет с искомым Ethernet-адресом, заносит его в свою таблицу трансляции, и отвечает на исходный запрос.

Конечно, может случиться и так, что ни одна машина не опознала свой IP-адрес в широковещательном запросе. В таком случае ARP отвечает, что машина с таким адресом недоступна в этой сети.

Протокол обратного разрешения адресов (RARP, Reverse Address Resolution Protocol) используется для перевода адресов Ethernet обратно в IP-адреса. Это нужно, например, когда хост загружает операционную систему с удалённой машины. Для этого он должен сначала узнать свой IP-адрес. Поэтому при загрузке он посылает широковещательный запрос вида «Скажите IP-адрес хоста с таким адресом Ehternet». RARP-сервер(если он вообще есть в этой сети и «слушает» запросы) должен в ответ послать IP-адрес.

8.3.4 Транспортный уровень

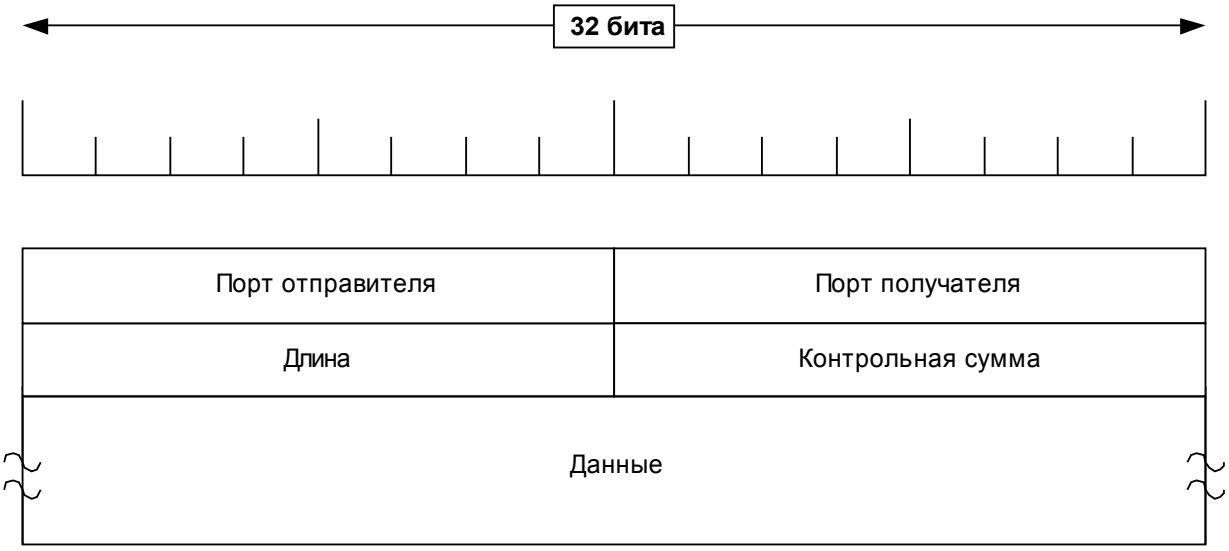
Над межсетевым уровнем находится транспортный уровень. Протоколы этого уровня пользуются для передачи данных протоколом IP и, как правило, сами используются прикладными программами. Два очень важных протокола транспортного уровня будут в центре внимания этой главы: Протокол контроля передачи и(TCP, Transmission Control Protocol) и Протокол пользовательских дейтаграмм (UDP, User Datagram Protocol). TCP

обеспечивает надежную доставку данных с автоматическим контролем и исправлением ошибок. UDP обеспечивает быструю доставку дейтаграмм , но без логического соединения.

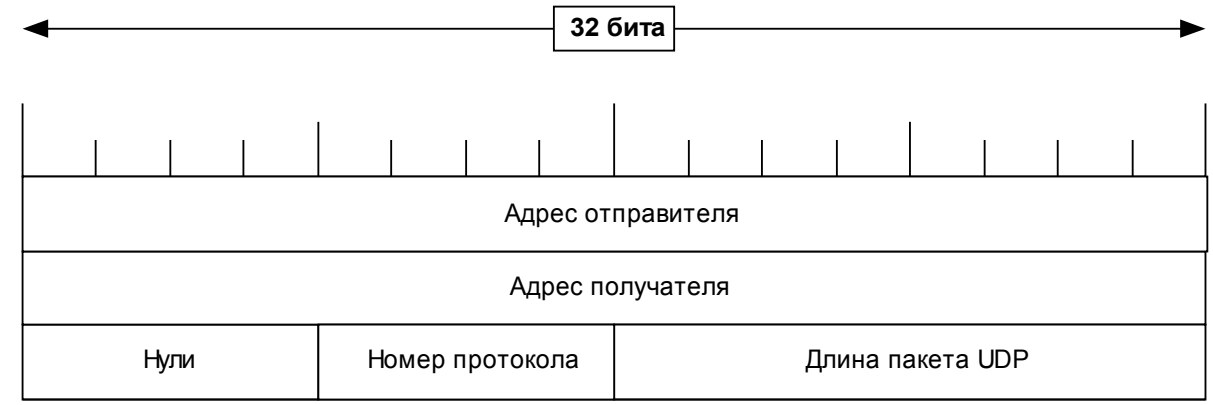
8.3.4.1 Протокол UDP

Этот протокол дает прикладным программам прямой доступ к сервису передачи дейтаграм, похожему на сервис, предоставляемый IP. Это позволяет программам обмениваться сообщениями с минимальной загрузкой сети.

UDP – это ненадежный (в том же смысле, в каком ненадежен протокол IP, то есть UDP, то есть UDP не занимается обнаружением и исправлением ошибок) протокол без логического соединения. То есть для того, чтобы отправит пакет, протокол не связывается с адресатом. Внутри компьютера протокол доставит данные, конечно, без ошибок. UDP использует 16-битные номера портов отправителя и получателя для идентификации соответствующего процесса. На рисунке показан заголовок сообщения UDP.

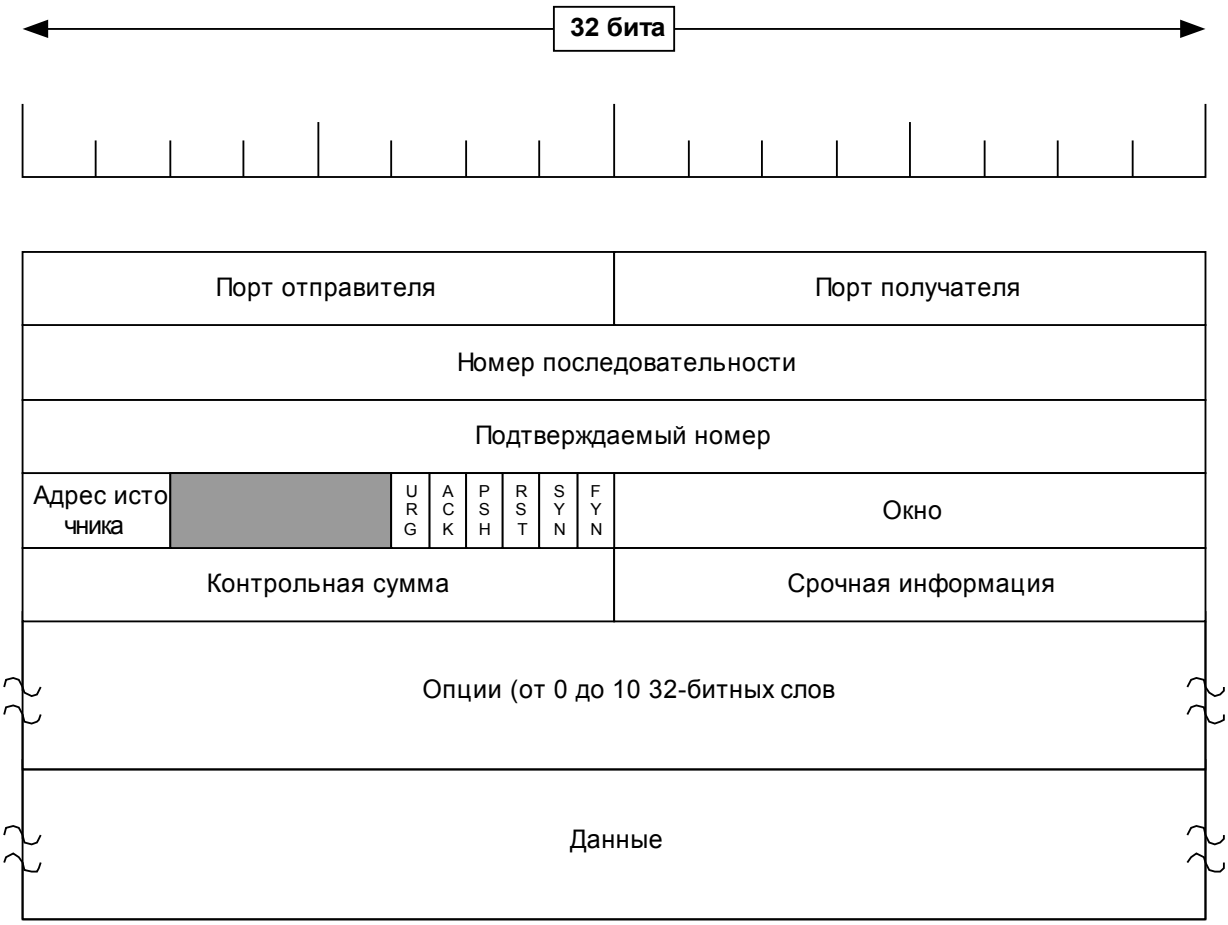


Здесь длина - это длина всего пакета UDP (всего пакета всего пакета UDP – данных вместе с заголовком. Контрольная сумма считается как поразрядное дополнение от псевдо-заголовка и пакета (вместе с данными). Кстати, нужно сказать, что такое псевдо-заголовок. Он состоит из адреса отправителя, адреса получателя номера протокола (для UDP это 17) и длины сообщения UDP (не выровненного по двум октетам). Его структура:

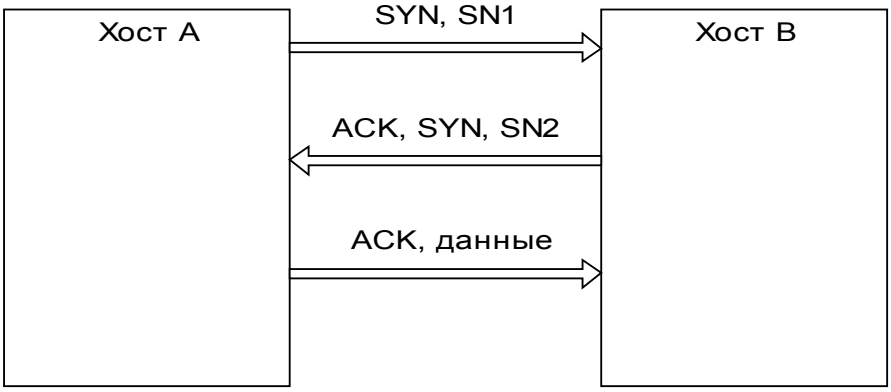


8.3.4.2 Протокол TCP

Программам, которым нужна надежная доставка данных, используют TCP. Этот протокол проверяет, что данные доставлены аккуратно и в правильной последовательности. TCP характеризуется тремя свойствами - это надежный протокол, реализующий связь с логическим соединением, который рассматривает данные как непрерывный поток байт. Давайте более подробно рассмотрим каждое из этих свойств: надежность, связь с логическим соединением, данные как непрерывный поток байтов.



TCP обеспечивает надежность с помощью механизма с хитроумным названием – Положительное подтверждение с повторной передачей (PAR, Positive Acknowledgment with Retransmission). Проще говоря, система, пользующаяся PAR, передает данные снова и снова до тех пор пока от удаленной машины не придет подтверждение о их корректной доставке. Единица данных, которой обмениваются системы с протоколом TCP называется сегментом. Каждый сегмент содержит контрольную сумму, по которой Получатель может проверить корректность доставки данных. Если данные дошли нормально, то он посылает отправителю положительное подтверждение. Если нет – данные просто игнорируются, а отправитель через некоторое время повторно посылает тот сегмент данных, о получении которого не было подтверждения.



TCP поддерживает связь с логическим соединением. Хосты обмениваются контрольной информацией, называемой оповещением (handshaking), чтобы установить контакт до передачи информации. TCP помечает сегмент как контрольный установкой соответствующего бита флагов в четвертом слове заголовка.

Способ установления соединения, используемый в TCP, называется тройным оповещением (three-way handshake), потому что для этого производится обмен тремя сегментами. На рисунке показан простейший пример применения такого способа. Хост А инициирует соединение передачей хосту В сегмента с установленным битом “ Синхронизация Номеров последовательностей” (SYN, “Synchronize sequence numbers”). Этот сегмент говорит хосту В о том, что А хочет установить соединение и будет использовать какой-то номер последовательности (Sequence Number) SN1 как стартовый для своих сегментов. (номера последовательностей используются для установления порядка в передаваемых данных). Хост В отвечает хосту А сегментом с установленными битами “ Подтверждение” (ACK, Acknowledgment) и SYN. Этим сегментом В подтверждает получение сегмента от А и информирует, с какого номера последовательности SN2 будет начинать свою передачу. Наконец, хост А передает сегмент, которым подтверждает получение сегмента от В, и в котором содержатся первые реальные данные.

После такого обмена протокол TCP хоста А уверен, что удаленный протокол работает и готов к получению данных. Как только устанавливается это соединение, можно обмениваться данными. Когда протоколы решат завершить обмен, они опять обмениваются сегментами с установленным биом “Нет больше данных для передачи” (FIN, “No more data from sender”). Таким образом они закрывают соединение. Весь этот обмен (до и после передачи данных) необходим для логического соединения между двумя системами.

TCP рассматривает передаваемые данные как непрерывный поток байт, а не как набор независимых сегментов. Поэтому ему важно знать порядок, в котором данные отправляются и принимаются.

8.3.4.3 Сравнение протоколов TCP и UDP

Зачем в TCP/IP два базовых протокола транспортного уровня?
Дело в том, что есть довольно большая разница между этими двумя протоколами – TCP и UDP. TCP обеспечивает надежную доставку данных с обнаружением и исправлением ошибок и с установлением логического соединения. UDP же ничего этого не делает, он просто отправляет пакеты с данными, не заботясь об их доставке. Зато загрузка сети становится меньше из-за отсутствия логического соединения.

8.3.5 Уровень прикладных программ

На самом верху архитектуры семейства протоколов TCP находится уровень прикладных программ. Все процессы этого уровня пользуются протоколами транспортного уровня для обмена данными по сети. Существует много протоколов прикладных программ. Рассмотрим на примере простого шлюза архитектуру всего TCP/IP и, заодно, несколько протоколов уровня прикладных программ. Рассмотрим, в качестве примеров, некоторые из известных прикладных протоколов.

8.3.5.1 Протоколы, опирающиеся на TCP

TELNET (Network Terminal Protocol), Протокол сетевого терминала, разработан для удаленного доступа к компьютерам сети (remote login).

FTP (File Transfer Protocol), Протокол передачи файлов, используется для интерактивной передачи файлов между компьютерами сети.

SMTP (Simple Mail Transfer Protocol), Простой протокол передачи почты. Основной протокол для обмена почтой, использующийся в Internet.

8.3.5.2 Протоколы, опирающиеся на UDP

DNS (Domain Name Service), Служба имен доменов, или просто Служба именования. С помощью этого протокола устанавливается взаимно однозначное соответствие между IP-адресами сетевых устройств и их именами. Об этом протоколе и о системе именования вообще речь пойдет дальше.

RIP (Routing Information Protocol), Протокол информации о маршрутизации. Маршрутизация данных (поиск путей их передачи от хоста-отправителя к хосту-получателю) в Internet является одной из важнейших функций семейства протоколов

TCP/IP. RIP используется сетевыми устройствами для обмена информацией о маршрутизации. Маршрутизация тоже будет подробно рассмотрена дальше.

NFS (Network File System), сетевая файловая система. С помощью этого протокола компьютеры могут совместно использовать файлы, разбросанные по сети.

9 Средства взаимодействия процессов в сети.

9.1 Сокеты.

Средства межпроцессного взаимодействия ОС UNIX, представленные в системе IPC, решают проблему взаимодействия двух процессов, выполняющихся в рамках одной операционной системы. Однако, очевидно, их невозможно использовать, когда требуется организовать взаимодействие процессов в рамках сети. Это связано как с принятой системой именования, которая обеспечивает уникальность только в рамках данной системы, так и вообще с реализацией механизмов разделяемой памяти, очереди сообщений и семафоров, – очевидно, что для удаленного взаимодействия они не годятся. Следовательно, возникает необходимость в каком-то дополнительном механизме, позволяющем общаться двум процессам в рамках сети. Однако если разработчики программ будут иметь два абсолютно разных подхода к реализации взаимодействия процессов, в зависимости от того, на одной машине они выполняются или на разных узлах сети, им, очевидно, придется во многих случаях создавать два принципиально разных куска кода, отвечающих за это взаимодействие. Понятно, что это неудобно и хотелось бы в связи с этим иметь некоторый унифицированный механизм, который в определенной степени позволял бы абстрагироваться от расположения процессов и давал бы возможность использования одних и тех же подходов для локального и нелокального взаимодействия. Кроме того, как только мы обращаемся к сетевому взаимодействию, встает проблема многообразия сетевых протоколов и их использования. Понятно, что было бы удобно иметь какой-нибудь общий интерфейс, позволяющий пользоваться услугами различных протоколов по выбору пользователя.

Обозначенные проблемы был призван решить механизм, впервые появившийся в Берклиевском UNIX – BSD, начиная с версии 4.2, и названный сокетами (**sockets**). Ниже подробно рассмотривается этот механизм.

Сокеты представляют собой в определенном смысле обобщение механизма каналов, но с учетом возможных особенностей, возникающих при работе в сети. Кроме того, они предоставляют по сравнению с каналами больше возможностей по передаче сообщений, например, могут поддерживать передачу экстренных сообщений вне общего потока данных. Общая схема работы с сокетами любого типа такова: каждый из взаимодействующих процессов должен на своей стороне создать и отконфигурировать сокет, после чего они должны осуществить соединение с использованием этой пары сокетов. По окончании взаимодействия сокеты уничтожаются.

Механизм сокетов чрезвычайно удобен при разработке взаимодействующих приложений, образующих систему «клиент-сервер». Клиент посылает серверу запросы на предоставление услуги, а сервер отвечает на эти запросы. Схема использования механизма сокетов для взаимодействия в рамках модели «клиент-сервер» такова. Процесс-сервер запрашивает у ОС сокет и, получив его, присваивает ему некоторое имя (адрес), которое предполагается заранее известным всем клиентам, которые захотят общаться с данным сервером. После этого сервер переходит в режим ожидания и обработки запросов от клиентов. Клиент, со своей стороны, тоже создает сокет и запрашивает соединение своего сокета с сокетом сервера, имеющим известное ему имя (адрес). После того, как соединение будет установлено, клиент и сервер могут обмениваться данными через соединенную пару сокетов. Ниже мы подробно рассмотрим функции, выполняющие все необходимые действия с сокетами, и напишем пример небольшой серверной и клиентской программы, использующих сокеты.

9.1.1 Типы сокетов. Коммуникационный домен.

Сокеты подразделяются на несколько типов в зависимости от типа коммуникационного соединения, который они используют. Два основных типа коммуникационных соединений и, соответственно, сокетов представляет собой соединение с использованием виртуального канала и датаграммное соединение.

Соединение с использованием виртуального канала – это последовательный поток байтов, гарантирующий надежную доставку сообщений с сохранением порядка их следования. Данные начинают передаваться только после того, как виртуальный канал установлен, и канал не разрывается, пока все данные не будут переданы. Примером соединения с установлением виртуального канала является механизм каналов в UNIX, аналогом такого соединения из реальной жизни также является телефонный разговор. Заметим, что границы сообщений при таком виде соединений не сохраняются, т.е. приложение, получающее данные, должно само определять, где заканчивается одно сообщение и начинается следующее. Такой тип соединения может также поддерживать передачу экстренных сообщений вне основного потока данных, если это возможно при использовании конкретного выбранного протокола.

Датаграммное соединение используется для передачи отдельных пакетов, содержащих порции данных – датаграмм. Для датаграмм не гарантируется доставка в том же порядке, в каком они были посланы. Вообще говоря, для них не гарантируется доставка вообще, надежность соединения в этом случае ниже, чем при установлении виртуального канала. Однако датаграммные соединения, как правило, более быстрые. Примером датаграммного соединения из реальной жизни может служить обычная почта: письма и посылки могут приходить адресату не в том порядке, в каком они были посланы, а некоторые из них могут и совсем пропадать.

Поскольку сокеты могут использоваться как для локального, так и для удаленного взаимодействия, встает вопрос о пространстве адресов сокетов. При создании сокета указывается т.н. *коммуникационный домен*, к которому данный сокет будет принадлежать. Коммуникационный домен определяет форматы адресов и правила их интерпретации. Мы будем рассматривать два основных домена: для локального взаимодействия – домен **AF_UNIX** и для взаимодействия в рамках сети – домен **AF_INET** (префикс AF обозначает сокращение от address family – семейство адресов). В домене **AF_UNIX** формат адреса – это допустимое имя файла, в домене **AF_INET** адрес образуют имя хоста + номер порта.

Заметим, что фактически коммуникационный домен определяет также используемые семейства протоколов. Так, для домена **AF_UNIX** это будут внутренние протоколы ОС, для домена **AF_INET** – протоколы семейства **TCP/IP. BSD UNIX** поддерживает также третий домен – **AF_NS**, использующий протоколы удаленного взаимодействия **Xerox NS**, но мы его рассматривать не будем.

Ниже приведен набор функций для работы с сокетами.

9.1.2 Создание сокета.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol)
```

Функция создания сокета так и называется – **`socket().`** У нее имеется три аргумента. Первый аргумент – **`domain`** – обозначает коммуникационный домен, к которому должен принадлежать создаваемый сокет. Для двух рассмотренных нами доменов соответствующие константы будут равны, как мы уже говорили, **AF_UNIX** и **AF_INET**. Второй аргумент – **`type`** – определяет тип соединения, которым будет пользоваться сокет (и, соответственно, тип сокета). Для двух основных рассматриваемых нами типов сокетов это будут константы **SOCK_STREAM** для соединения с установлением виртуального канала и **SOCK_DGRAM**

для датаграмм[4]. Третий аргумент – **protocol** – задает конкретный протокол, который будет использоваться в рамках данного коммуникационного домена для создания соединения. Если установить значение данного аргумента в 0, система автоматически выберет подходящий протокол. В наших примерах мы так и будем поступать. Однако здесь для справки приведем константы для протоколов, используемых в домене **AF_INET:**

IPPROTO_TCP – обозначает протокол **TCP** (корректно при создании сокета типа **SOCK_STREAM**)

IPPROTO_UDP – обозначает протокол **UDP** (корректно при создании сокета типа **SOCK_DGRAM**)

Функция socket возвращает в случае успеха положительное целое число – дескриптор сокета, аналог файлового дескриптора, которое может быть использовано в дальнейших вызовах при работе с данным сокетом. В случае если создание сокета с указанными параметрами невозможно (например, при некорректном сочетании коммуникационного домена, типа сокета и протокола), функция возвращает **–1**.

9.1.3 Связывание.

Для того чтобы к созданному сокету мог обратиться какой-либо процесс извне, необходимо присвоить ему адрес. Как мы уже говорили, формат адреса зависит от коммуникационного домена, в рамках которого действует сокет, и может представлять собой либо путь к файлу, либо сочетание IP-адреса и номера порта. Но в любом случае присвоение связывание сокета с конкретным адресом осуществляется одной и той же функцией `bind`:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind (int sockfd, struct sockaddr *myaddr, int addrlen)
```

Первый аргумент функции – дескриптор сокета, возвращенный функцией **socket()**; второй аргумент – указатель на структуру, содержащую адрес сокета. Для домена **AF_UNIX** формат структуры описан в **<sys/un.h>** и выглядит следующим образом:

```
#include <sys/un.h>
struct sockaddr_un {
      short sun_family; /* == AF_UNIX */
      char sun_path[108];
};
```

Для домена **AF_INET** формат структуры описан в **<netinet/in.h>** и выглядит следующим образом:

```
#include <netinet/in.h>
struct sockaddr_in {
      short sin_family; /* == AF_INET */
      u_short sin_port;  /* port number */
      struct in_addr sin_addr;  /* host IP address */
      char sin_zero[8];  /* not used */
};
```

Последний аргумент функции задает реальный размер структуры, на которую указывает **myaddr.** Отметим, что если мы имеем дело с локальными сокетами и адрес сокета представляет собой имя файла, то при выполнении функции **bind** система в качестве побочного эффекта создает файл с таким именем. Поэтому для успешного выполнения `bind` необходимо, чтобы такого файла не существовало к данному моменту. Это следует

---

[4] Заметим, что данный аргумент может принимать не только указанные два значения, например, тип сокета **SOCK_SEQPACKET** обозначает соединение с установлением виртуального канала со всеми вытекающими отсюда свойствами, но при этом сохраняются границы сообщений; однако данный тип сокетов не поддерживается ни в домене **AF_UNIX**, ни в домене **AF_INET**, поэтому мы его здесь рассматривать не будем

учитывать, если мы «зашиваем» в программу определенное имя и намерены запускать нашу программу несколько раз – необходимо удалять этот файл перед связыванием.

В случае успешного связывания **bind** возвращает **0**, в случае ошибки – **-1.**

Сокеты с предварительным установлением соединения. Запрос на соединение.

Различают сокеты с предварительным установлением соединения, когда до начала передачи данных устанавливаются адреса сокетов отправителя и получателя данных – сокеты соединяются друг с другом и остаются соединенными до окончания обмена данными и сокеты без установления соединения, когда соединение до начала передачи данных не устанавливается, а адреса сокетов отправителя и получателя передаются с каждым сообщением. Если тип сокета –виртуальный канал, то сокет ***должен*** устанавливать соединение, если же тип сокета – датаграмма, то, как правило, это сокет без установления соединения, хотя последнее не является требованием. Для установления соединения служит следующая функция:

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Здесь первый аргумент – дескриптор сокета, второй аргумент – указатель на структуру, содержащую адрес сокета, с которым производится соединение, в формате, который мы обсуждали выше, и третий аргумент содержит реальную длину этой структуры. Функция возвращает **0** в случае успеха и **–1** в случае неудачи, при этом код ошибки можно посмотреть в переменной `errno`.

Заметим, что в рамках модели «клиент-сервер» клиенту, вообще говоря, не важно, какой адрес будет назначен сокету, т.к. никакой процесс не будет пытаться непосредственно установить соединение с сокетом клиента. Поэтому клиент может не вызывать предварительно функцию `bind`, в этом случае при вызове `connect` система автоматически выберет приемлемые значения для локального адреса клиента. Однако сказанное справедливо только для взаимодействия в рамках домена **AF_INET**, в домене **AF_UNIX** клиентское приложение само должно позаботиться о связывании сокета.

9.1.4 Сервер: прослушивание сокета и подтверждение соединения.

Следующие два вызова используются сервером только в том случае, если используются сокеты с предварительным установлением соединения.

```
#include <sys/types.h>
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

Этот вызов используется процессом-сервером для того, чтобы сообщить системе о том, что он готов к обработке запросов на соединение, поступающих на данный сокет. До тех пор, пока процесс – владелец сокета не вызовет **listen,** все запросы на соединение с данным сокетом будут возвращать ошибку. Первый аргумент функции – дескриптор сокета. Второй аргумент, **backlog,** содержит максимальный размер очереди запросов на соединение. ОС буферизует приходящие запросы на соединение, выстраивая их в очередь до тех пор, пока процесс не сможет их обработать. В случае если очередь запросов на соединение переполняется, поведение ОС зависит от того, какой протокол используется для соединения. Если конкретный протокол соединения не поддерживает возможность перепосылки (retransmission) данных, то соответствующий вызов **connect** вернет ошибку **ECONNREFUSED**. Если же перепосылка поддерживается (как, например, при использовании **TCP**), ОС просто выбрасывает пакет, содержащий запрос на соединение, как если бы она его не получала вовсе. При этом пакет будет присылаться повторно до тех пор, пока очередь запросов не уменьшится и попытка соединения не увенчается успехом, либо пока не произойдет тайм-аут, определенный для протокола. В последнем случае вызов `connect` завершится с ошибкой **ETIMEDOUT**. Это позволит клиенту отличить, был ли

процесс-сервер слишком занят, либо он не функционировал. В большинстве систем максимальный допустимый размер очереди равен 5.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *addr, int *addrlen);
```

Этот вызов применяется сервером для удовлетворения поступившего клиентского запроса на соединение с сокетом, который сервер к тому моменту уже прослушивает (т.е. предварительно была вызвана функция **`listen`**). **`Accept`** извлекает первый запрос из очереди и устанавливает с ним соединение. Если к моменту вызова **accept** никаких запросов на соединение с данным сокетом еще не поступало, процесс, вызвавший **`accept`**, блокируется до поступления запросов. Когда запрос поступает и соединение устанавливается, **`accept`** возвращает дескриптор нового сокета, соединенного с сокетом клиентского процесса. Через этот новый сокет и осуществляется обмен данными, в то время как старый сокет продолжает обрабатывать другие поступающие запросы на соединение (напомним, что именно первоначально созданный сокет связан с адресом, известным клиентам, поэтому все клиенты могут слать запросы только на соединение с этим сокетом). Это позволяет процессу-серверу поддерживать несколько соединений одновременно. Обычно это реализуется путем порождения для каждого установленного соединения отдельного процесса-потомка, который занимается собственно обменом данными только с этим конкретным клиентом, в то время как процесс-родитель продолжает прослушивать первоначальный сокет и порождать новые соединения. Во втором параметре передается указатель на структуру, в которой возвращается адрес клиентского сокета, с которым установлено соединение, а в третьем параметре возвращается реальная длина этой структуры. Благодаря этому сервер всегда знает, куда ему в случае надобности следует послать ответное сообщение. Если адрес клиента нас не интересует, в качестве второго аргумента можно передать **NULL**.

9.1.5 Прием и передача данных.

Собственно для приема и передачи данных через сокет используются три пары функций.

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sockfd, const void *msg, int len, unsigned int flags);
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

Эти функции используются для обмена только через сокет с предварительно установленным соединением. Аргументы функции **`send:`** **`sockfd`** – дескриптор сокета, через который передаются данные, **`msg`** и `len` - сообщение и его длина. Если сообщение слишком длинное для того протокола, который используется при соединении, оно не передается и вызов возвращает ошибку **EMSGSIZE.** Если же сокет окажется переполнен, т.е. в его буфере не хватит места, чтобы поместить туда сообщение, выполнение процесса блокируется до появления возможности поместить сообщение. Функция `send` возвращает количество переданных байт в случае успеха и -1 в случае неудачи. Код ошибки при этом устанавливается в `errno`. Аргументы функции `recv` аналогичны: **`sockfd`** – дескриптор сокета, **`buf`** и **`len`** – указатель на буфер для приема данных и его первоначальная длина. В случае успеха функция возвращает количество считанных байт, в случае неудачи -1[5].

[5] Отметим, что, как уже говорилось, при использовании сокетов с установлением виртуального соединения границы сообщений не сохраняются, поэтому приложение, принимающее сообщения, может принимать данные совсем не теми же порциями, какими они были посланы. Вся работа по интерпретации сообщений возлагается на приложение.

Последний аргумент обеих функций – **flags** – может содержать комбинацию специальных опций. Нас будут интересовать две из них:

MSG_OOB - этот флаг сообщает ОС, что процесс хочет осуществить прием/передачу экстренных сообщений

MSG_PEEK – данный флаг может устанавливаться при вызове recv. При этом процесс получает возможность прочитать порцию данных, не удаляя ее из сокета, таким образом, что последующий вызов recv вновь вернет те же самые данные.

Другая пара функций, которые могут использоваться при работе с сокетами с предварительно установленным соединением – это обычные **read()** и **write()**, в качестве дескриптора которым передается дескриптор сокета.

И, наконец, пара функций, которая может быть использована как с сокетами с установлением соединения, так и с сокетами без установления соединения:

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);
```

Первые 4 аргумента у них такие же, как и у рассмотренных выше. В последних двух в функцию sendto должны быть переданы указатель на структуру, содержащую адрес получателя, и ее размер, а функция **recvfrom** в них возвращает соответственно указатель на структуру с адресом отправителя и ее реальный размер. Отметим, что перед вызовом recvfrom параметр **fromlen** должен быть установлен равным первоначальному размеру структуры **from.** Здесь, как и в функции **accept**, если нас не интересует адрес отправителя, в качестве **from** можно передать **NULL.**

9.1.6 Закрытие сокета.

Если процесс закончил прием либо передачу данных, ему следует закрыть соединение. Это можно сделать с помощью функции shutdown**:**

```
# include <sys/types.h>
# include <sys/socket.h>
int shutdown (int sockfd, int mode);
```

Помимо дескриптора сокета, ей передается целое число, которое определяет, какую режим закрытия соединения. Если **mode=0,** сокет закрывается для чтения, при этом все дальнейшие попытки чтения будут возвращать **end-of-file**. Если **mode=1**, то сокет закрывается для записи, и дальнейшие попытки передать данные вернут ошибку (-1). Если **mode=2**, то сокет закрывается и для чтения, и для записи. В принципе, для закрытия сокета можно было бы воспользоваться просто функцией close, но тут есть одно отличие. Если используемый для соединения протокол гарантирует доставку данных (тип сокета – виртуальный канал), то вызов **close** будет блокирован до тех пор, пока система будет пытаться доставить все данные, находящиеся «в пути», в то время как вызов **shutdown** извещает систему о том, что эти данные уже не нужны и можно не предпринимать попыток их доставить.

9.1.7 Базовые схемы организации использования.

Мы рассмотрели все основные функции работы с сокетами. Обобщая изложенное, можно изобразить общую схему работы с сокетами с установлением соединения в следующем виде:

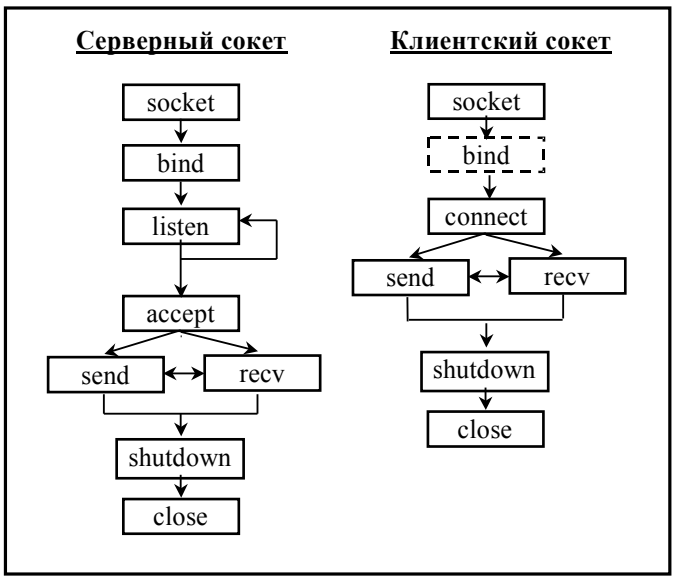


Рис. 8Схема работы с сокетами с установлением соединения

Общая схема работы с сокетами без предварительного установления соединения проще, она такова:

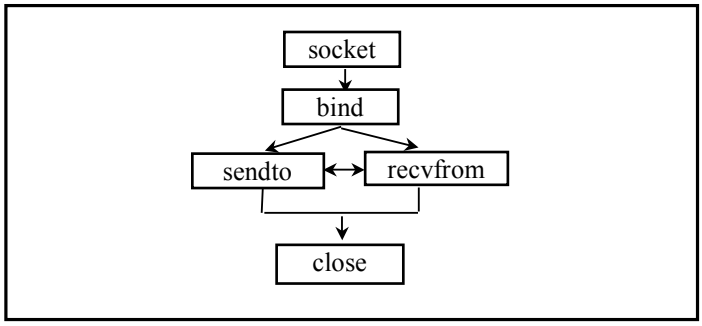


Рис. 9 Схема работы с сокетами без установления соединения

Пример. Работа с локальными сокетами.

Рассмотрим небольшой пример, иллюстрирующий работу с сокетами в рамках локального домена **(AF_UNIX).** Ниже приведена небольшая программа, которая в зависимости от параметра командной строки исполняет роль клиента или сервера. Клиент и сервер устанавливают соединение с использованием датаграммных сокетов. Клиент читает строку со стандартного ввода и пересылает серверу; сервер посылает ответ в зависимости от того, какова была строка. При введении строки «quit» и клиент, и сервер завершаются.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#include <string.h>

#define SADDRESS "mysocket"
#define CADDRESS "clientsocket"
#define BUFLEN  40
int main(int argc, char **argv)
{
      struct sockaddr_un party_addr, own_addr;
      int sockfd;
      int is_server;
      char buf[BUFLEN];
      int party_len;
      int quitting;
```

```
      if (argc != 2) {
        printf("Usage: %s client|server.\n", argv[0]);
        return 0;
      }
      quitting = 1;

      /* определяем, кто мы: клиент или сервер*/
      is_server = !strcmp(argv[1], "server");
      memset(&own_addr, 0, sizeof(own_addr));
      own_addr.sun_family = AF_UNIX;
      strcpy(own_addr.sun_path, is_server?SADDRESS:CADDRESS);

      /* создаем сокет */
      if ((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
        printf("can't create socket\n");
        return 0;
      }

/* связываем сокет */
      unlink(own_addr.sun_path);
      if (bind(sockfd, (struct sockaddr *) &own_addr,
          sizeof(own_addr.sun_family)+
          strlen(own_addr.sun_path)) < 0)
      {
            printf("can't bind socket!");
            return 0;
      }

      if (!is_server)
      {
            /* это - клиент */
            memset(&party_addr, 0, sizeof(party_addr));
            party_addr.sun_family = AF_UNIX;
            strcpy(party_addr.sun_path, SADDRESS);
            printf("type the string: ");

            while (gets(buf)) {
                  /* не пора ли выходить? */
                  quitting = (!strcmp(buf, "quit"));

                  /* считали строку и передаем ее серверу */
                  if (sendto(sockfd, buf, strlen(buf) + 1, 0,
                        (struct sockaddr *) &party_addr,
                        sizeof(party_addr.sun_family) +
                        strlen(SADDRESS)) != strlen(buf) + 1)
                  {
                        printf("client: error writing socket!\n");
                        return 0;
                  }

             /*получаем ответ и выводим его на печать*/
                  if (recvfrom(sockfd, buf, BUFLEN, 0, NULL, 0) < 0)
                  {
                        printf("client: error reading socket!\n");
                        return 0;
                  }
                  printf("client: server answered: %s\n", buf);
```

```
                if (quitting) break;
                printf("type the string: ");
          } // while
          close(sockfd);
          return 0;
     } // if (!is_server)

     /* это – сервер */
     while (1)
     {
          /* получаем строку от клиента и выводим на печать */
          party_len = sizeof(party_addr);
          if (recvfrom(sockfd, buf, BUFLEN, 0,(struct sockaddr *)
                    &party_addr, &party_len) < 0)
          {
                    printf("server: error reading socket!");
                    return 0;
          }

          printf("server: received from client: %s \n", buf);
          /* не пора ли выходить? */
          quitting = (!strcmp(buf, "quit"));
          if (quitting) strcpy(buf, "quitting now!");
          else
                if (!strcmp(buf, "ping!")) strcpy(buf, "pong!");
                else strcpy(buf, "wrong string!");
          /* посылаем ответ */
          if (sendto(sockfd, buf, strlen(buf) + 1, 0,
                (struct sockaddr *) &party_addr,
                party_len) != strlen(buf)+1)
          {
                printf("server: error writing socket!\n");
                return 0;
          }
          if (quitting) break;
     } // while
     close(sockfd);
     return 0;
}
```

Пример работы с сокетами в рамках сети.

В качестве примера работы с сокетами в домене **AF_INET** напишем простой web-сервер, который будет понимать только одну команду :

GET /<имя файла>

Сервер запрашивает у системы сокет, связывает его с адресом, считающимся известным, и начинает принимать клиентские запросы. Для обработки каждого запроса порождается отдельный потомок, в то время как родительский процесс продолжает прослушивать сокет. Потомок разбирает текст запроса и отсылает клиенту либо содержимое требуемого файла, либо диагностику (“плохой запрос” или “файл не найден”).

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
```

```
#include <unistd.h>

#define PORTNUM 8080
#define BACKLOG 5
#define BUFLEN 80

#define FNFSTR "404 Error File Not Found "
#define BRSTR "Bad Request "

int main(int argc, char **argv)
{
      struct sockaddr_in own_addr, party_addr;
      int sockfd, newsockfd, filefd;
      int party_len;
      char buf[BUFLEN];
      int len;
      int i;
      /* создаем сокет */
      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            printf("can't create socket\n");
            return 0;
      }
      /* связываем сокет */
      memset(&own_addr, 0, sizeof(own_addr));
      own_addr.sin_family = AF_INET;
      own_addr.sin_addr.s_addr = INADDR_ANY;
      own_addr.sin_port = htons(PORTNUM);
if (bind(sockfd, (struct sockaddr *) &own_addr,
            sizeof(own_addr)) < 0)
      {
            printf("can't bind socket!");
            return 0;
      }

      /* начинаем обработку запросов на соединение */
      if (listen(sockfd, BACKLOG) < 0)
      {
            printf("can't listen socket!");
            return 0;
      }

      while (1) {
            memset(&party_addr, 0, sizeof(party_addr));
            party_len = sizeof(party_addr);
            /* создаем соединение */
            if ((newsockfd = accept(sockfd, (struct sockaddr *)
                  &party_addr, &party_len)) < 0)
            {
                  printf("error accepting connection!");
                  return 0;
            }

            if (!fork())
            {
                  /*это – сын, он обрабатывает запрос и посылает
                  ответ*/
                  close(sockfd);   /*  этот сокет сыну не нужен */
```

```
                    if ((len = recv(newsockfd, &buf, BUFLEN, 0)) < 0)
                    {
                         printf("error reading socket!");
                         return 0;
                    }
                    /* разбираем текст запроса */
                    printf("received: %s \n", buf);
                    if (strncmp(buf, "GET /", 5))
                    { /*плохой запрос!*/
                         if (send(newsockfd, BRSTR, strlen(BRSTR) + 1,
                    0) != strlen(BRSTR) + 1)
                         {
                              printf("error writing socket!");
                              return 0;
                         }

                         shutdown(newsockfd, 1);
                         close(newsockfd);
                         return 0;
                    }

                    for (i=5;buf[i] && (buf[i] > ' ');i++);
                    buf[i] = 0;
                    /* открываем файл */
                    if((filefd = open(buf+5, O_RDONLY)) < 0) {
                         /* нет файла! */
                         if (send(newsockfd, FNFSTR,
     strlen(FNFSTR) + 1, 0) != strlen(FNFSTR) + 1)
                         {
                              printf("error writing socket!");
                              return 0;
                         }
                         shutdown(newsockfd, 1);
                         close(newsockfd);
                         return 0;
                    }

                    /* читаем из файла порции данных и посылаем их
клиенту */
                    while (len = read(filefd, &buf, BUFLEN))
                    if (send(newsockfd, buf, len, 0) < 0) {
                         printf("error writing socket!");
                         return 0;
                    }
                    close(filefd);
                    shutdown(newsockfd, 1);
                    close(newsockfd);
                    return 0;
               }

          /* процесс – отец. Он закрывает новый сокет и
          продолжает прослушивать старый */
          close(newsockfd);
     }
}
```

9.2 Интерфейс передачи сообщений: MPI.

9.2.1 Базовые архитектуры.

Основным параметром классификации параллельных компьютеров является наличие общей памяти (в симметричных мультипроцессорных системах – SMP) или распределенной памяти (в массивно-параллельных системах – MPP). Нечто среднее между SMP и MPP представляют собой NUMA архитектуры – системы с неоднородным доступом к памяти. Память в NUMA-системах физически распределена, но логически общедоступна. Кластерные системы являются более дешевым вариантом MPP.

Массивно-параллельные системы (MPP) состоят из однородных вычислительных узлов, включающих

- один или несколько процессоров (обычно RISC – reduced instruction set computer),
- локальную память (прямой доступ к памяти с других узлов невозможен),
- коммуникационный процессор или сетевой адаптер,
- иногда – жесткие диски и/или другие устройства ввода/вывода.

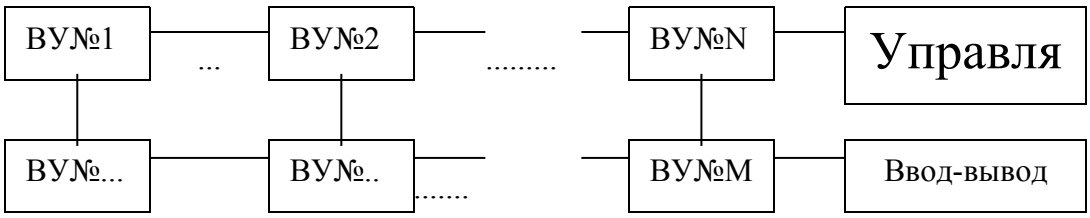
К системе могут быть добавлены специальные узлы ввода-вывода и управляющие узлы. Узлы связаны через высокоскоростную сеть, коммутатор и т.п. Число процессоров в MPP-системах может достигать нескольких тысяч.

Существует два способа управления массивно-параллельными системами.

1 На каждом узле может работать полноценная, UNIX-подобная операционная система, функционирующая отдельно от других узлов.

2 Полноценная ОС работает только на управляющей машине, на каждом узле работает сильно урезанный вариант ОС, обеспечивающий работу соответствующей ветви параллельного приложения.

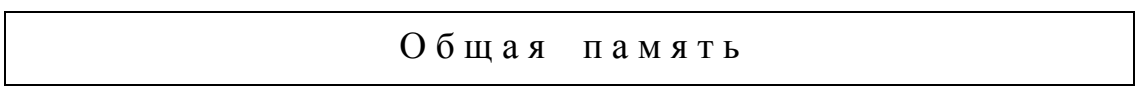
Схема MPP системы, где каждый вычислительный узел (ВУ) имеет процессорный элемент (например, RISC процессор, одинаковый для всех ВУ), память и коммуникационное устройство:

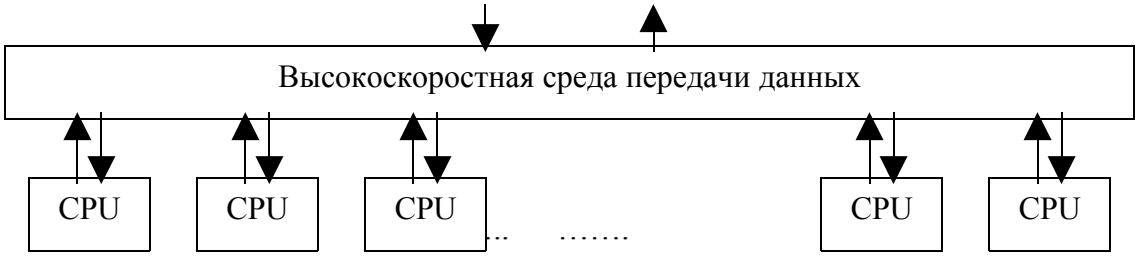


Программы для MPP-систем создаются в рамках модели передачи сообщений. Применяются технологии MPI, PVM и др.

Примеры массивно-параллельных систем – SGI/CRAY T3E, транспьютерные системы Parsytec.

Симметричные мультипроцессорные системы (SMP) состоят из нескольких **однородных** процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти либо с помощью общей шины, либо с помощью коммутатора. Аппаратно поддерживается согласованность данных в кэшах. Схематично SMP архитектуру можно изобразить так:





Наличие общей памяти сильно упрощает взаимодействие процессоров, однако накладывает сильные ограничения на их число - не более 32 в реальных системах. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры. Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически (в процессе работы) распределяет процессы/нити по процессорам, но иногда возможна и явная привязка. Программы для SMP-систем создаются в рамках модели общей памяти.. (POSIX threads, OpenMP). Стандарт MPI-2 позволяет создавать программы и для SMP-систем.

Системы с неоднородным доступом к памяти (NUMA) строятся на однородных базовых модулях. Модули состоят из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора. Поддерживается единое адресное пространство, аппаратно поддерживается доступ к удаленной памяти, т.е. к памяти других модулей. При этом доступ к локальной памяти в несколько раз быстрее, чем к удаленной. Обычно аппаратно поддерживается когерентность кэшей во всей системе (архитектура cc-NUMA – cache-coherent NUMA).

Масштабируемость NUMA-систем ограничивается объемом адресного пространства, возможностями аппаратуры поддержки когерентности кэшей и возможностями операционной системы по управлению большим числом процессоров.

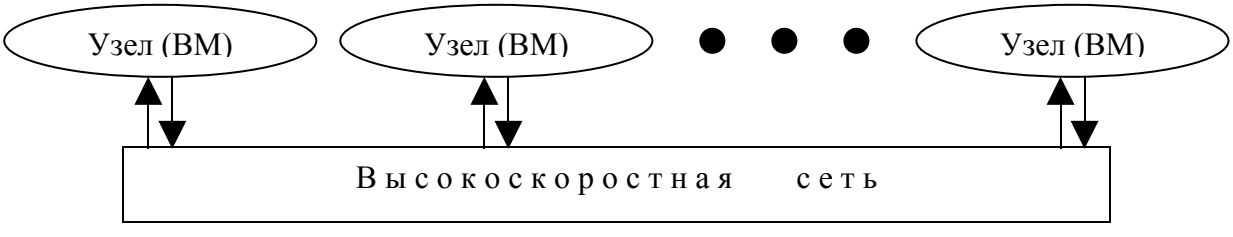
Обычно вся система работает под управлением единой ОС. Но возможны также варианты динамического "подразделения" системы, когда отдельные "разделы" системы работают под управлением разных ОС (например, Windows NT и UNIX в NUMA-Q 2000). Модель программирования - аналогична той, что используется на симметричных мультипроцессорных системах.

Примеры NUMA-систем: Sun HPC 10000, IBM/Sequent NUMA-Q 2000, SNI RM600.

Кластерные системы представляют собой набор рабочих станций (или даже ПК) общего назначения и используется в качестве дешевого варианта массивно-параллельного компьютера. Для связи узлов используется одна из стандартных сетевых технологий (Fast/Gigabit Ethernet, Myrinet). При объединении в кластер компьютеров разной мощности или разной архитектуры, говорят о гетерогенных (неоднородных) кластерах. Узлы кластера могут одновременно использоваться в качестве пользовательских рабочих станций.

В качестве операционной системы в кластерных системах чаще всего применяются свободно распространяемые ОС - Linux/FreeBSD, вместе со специальными средствами поддержки параллельного программирования и распределения нагрузки.

Взаимодействие узлов осуществляется, как правило, в рамках модели передачи сообщений (чаще всего - MPI). Дешевизна подобных систем оборачивается большими накладными расходами на взаимодействие параллельных процессов между собой, что сильно сужает потенциальный класс решаемых задач. Примеры: NT-кластер в NCSA, Beowulf-кластеры.



Обычно создание любой параллельной программы включает в себя следующие основных стадии:

- последовательный алгоритм подвергается декомпозиции (распараллеливанию), т.е. разбивается на независимо работающие ветви;
- для взаимодействия в ветви вводятся две дополнительных нематематических операции: прием и передача данных.
- распараллеленный алгоритм записывается в виде программы, в которой операции приема и передачи записываются в терминах конкретной системы связи между ветвями.

Система связи, в свою очередь, включает в себя два компонента: программный и аппаратный. С точки же зрения программиста данные могут передаваться:

- через разделяемую память, синхронизация доступа ветвей к такой памяти происходит посредством семафоров
- в виде сообщений

Первый метод является базовым для SMP-машин, второй - для сетей всех типов. Хотя можно реализовать любой метод на любой архитектуре, к примеру, разделяемая память на не-SMP-комплексах посредством организации единого виртуального адресного пространства или на SMP-машине вырожденным каналом связи для передачи сообщений служит разделяемая память.

9.2.2 Краткая характеристика MPI.

MPI представляет собой программный инструмент, предназначенных для поддержки работы параллельных процессов в терминах передачи сообщений для обеспечения связи между ветвями параллельного приложения. MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные / многопроцессорные с общей/раздельной памятью), взаимного расположения ветвей (на одном процессоре / на разных) и API операционной системы. Параллельное приложение состоит из нескольких ветвей, или процессов, или задач, выполняющихся одновременно. Разные процессы могут выполняться как на разных процессорах, так и на одном и том же - для программы это роли не играет, поскольку в обоих случаях механизм обмена данными одинаков. При программировании на MPI программа должна содержать код всех ветвей сразу. MPI-загрузчиком запускается указываемое количество экземпляров программы. Каждый экземпляр определяет свой порядковый номер в запущенном коллективе, и в зависимости от этого номера и размера коллектива выполняет ту или иную ветку алгоритма. Такая модель параллелизма называется *Single program/Multiple data* (SPMD), и является частным случаем модели *Multiple instruction/Multiple data* (MIMD). Каждая ветвь имеет пространство данных, полностью изолированное от других ветвей. Процессы обмениваются друг с другом данными в виде сообщений. Сообщения проходят под идентификаторами, которые позволяют программе и библиотеке связи отличать их друг от друга. Для совместного проведения тех или иных расчетов процессы внутри приложения объединяются в группы. Каждый процесс может узнать у библиотеки связи свой номер внутри группы, и, в зависимости от номера

приступает к выполнению соответствующей части расчетов. Количество ветвей фиксировано - в ходе работы порождение новых ветвей невозможно. Если MPI-приложение запускается в сети, запускаемый файл приложения должен быть построен на каждой машине.
В состав MPI входят, как правило, два обязательных компонента: библиотека программирования для языков Си, Си++ и Фортран, загрузчик исполняемых файлов. Кроме того, может присутствовать справочная система, командные файлы для облегчения компиляции/компоновки программ и др. в зависимости от версии.

9.2.3 Коммуникаторы, группы и области связи.

Группа - это некое множество ветвей. Одна ветвь может быть членом нескольких групп. В распоряжение программиста предоставлен тип **MPI_Group** и набор функций, работающих с переменными и константами этого типа. Констант, собственно, две: MPI_GROUP_EMPTY может быть возвращена, если группа с запрашиваемыми характеристиками в принципе может быть создана, но пока не содержит ни одной ветви; MPI_GROUP_NULL возвращается, когда запрашиваемые характеристики противоречивы. Согласно концепции MPI, после создания группу нельзя дополнить или усечь - можно создать только новую группу под требуемый набор ветвей на базе существующей.

Область связи ("communication domain") - это нечто абстрактное: в распоряжении программиста нет типа данных, описывающего непосредственно области связи, как нет и функций по управлению ими. Области связи автоматически создаются и уничтожаются вместе с коммуникаторами. Абонентами одной области связи являются ВСЕ задачи либо одной, либо двух групп.

Коммуникатор, или описатель области связи - это верхушка трехслойного пирога (группы, области связи, описатели областей связи), в который "запечены" задачи: именно с коммуникаторами программист имеет дело, вызывая функции пересылки данных, а также подавляющую часть вспомогательных функций. Одной области связи могут соответствовать несколько коммуникаторов. Коммуникаторы являются "несообщающимися сосудами": если данные отправлены через один коммуникатор, ветвь-получатель сможет принять их только через этот же самый коммуникатор, но ни через какой-либо другой.

Зачем вообще нужны разные группы, разные области связи и разные их описатели?
По существу, они служат той же цели, что и идентификаторы сообщений - помогают ветви-приемнику и ветви-получателю надежнее определять друг друга, а также содержимое сообщения. Ветви внутри параллельного приложения могут объединяться в подколлективы для решения промежуточных задач - посредством создания групп, и областей связи над группами. Пользуясь описателем этой области связи, ветви гарантированно ничего не примут извне подколлектива, и ничего не отправят наружу. Параллельно при этом они могут продолжать пользоваться любым другим имеющимся в их распоряжении коммуникатором для пересылок вне подколлектива, например, MPI_COMM_WORLD для обмена данными внутри всего приложения. Коллективные функции создают дубликат от полученного аргументом коммуникатора, и передают данные через дубликат, не опасаясь, что их сообщения будут случайно перепутаны с сообщениями функций "точка-точка", распространяемыми через оригинальный коммуникатор. Программист с этой же целью в разных кусках кода может передавать данные между ветвями через разные коммуникаторы, один из которых создан копированием другого. Коммуникаторы распределяются автоматически (функциями семейства "Создать новый комуникатор"), и для них не существует джокеров ("принимай через какой угодно коммуникатор") - вот еще два их существенных достоинства перед идентификаторами сообщений. Идентификаторы (целые числа) распределяются пользователем вручную, и это служит источником двух частых ошибок вследствие путаницы на приемной стороне:

- сообщениям, имеющим разный смысл, вручную по ошибке назначается один и тот же идентификатор;
- функция приема с джокером сгребает все подряд, в том числе и те сообщения, которые должны быть приняты и обработаны в другом месте ветви.

9.2.4 Обрамляющие функции. Начало и завершение.

Существует несколько функций, которые используются в любом, даже самом коротком приложении MPI. Занимаются они не столько собственно передачей данных, сколько ее обеспечением:

- Инициализация библиотеки. Одна из первых инструкций в функции main (главной функции приложения):

```
MPI_Init( &argc, &argv );
```

Она получает адреса аргументов, стандартно получаемых самой main от операционной системы и хранящих параметры командной строки. В конец командной строки программы MPI-загрузчик **mpirun** добавляет ряд информационных параметров, которые требуются MPI_Init. (Пример №1).

- Аварийное закрытие библиотеки. Вызывается, если пользовательская программа завершается по причине ошибок времени выполнения, связанных с MPI:

```
MPI_Abort( описатель области связи, код ошибки MPI );
```

Вызов MPI_Abort из любой задачи принудительно завершает работу ВСЕХ задач, подсоединенных к заданной области связи. Если указан описатель MPI_COMM_WORLD, будет завершено все приложение (все его задачи) целиком, что, по-видимому, и является наиболее правильным решением. Используйте код ошибки MPI_ERR_OTHER, если не знаете, как охарактеризовать ошибку в классификации MPI.

- Нормальное закрытие библиотеки:

```
MPI_Finalize();
```

Рекомендуется не забывать вписывать эту инструкцию перед возвращением из программы, то есть:

- перед вызовом стандартной функции Си **exit** ;
- перед каждым после MPI_Init оператором **return** в функции main ;
- если функции main назначен тип void, и она не заканчивается оператором return, то MPI_Finalize() следует поставить в конец main.

- Две информационных функции: сообщают размер группы (то есть общее количество задач, подсоединенных к ее области связи) и порядковый номер вызывающей задачи:

```
int size, rank;
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

Использование MPI_Init, MPI_Finalize, MPI_Comm_size и MPI_Comm_rank. – пример:

```
/*
 *  Начало и завершение:
 *      MPI_Init, MPI_Finalize
 *  Определение количества задач в приложениии и своего порядкового номера:
 *      MPI_Comm_size, MPI_Comm_rank
 */
#include <mpi.h>
#include <stdio.h>

int main( int argc, char **argv )
```

```
{
   int size, rank, i;

    /* Инициализируем библиотеку */
   MPI_Init( &argc, &argv );

    /* Узнаем количество задач в запущенном приложении */
   MPI_Comm_size( MPI_COMM_WORLD, &size );

    /* ... и свой собственный номер: от 0 до (size-1) */
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* задача с номером 0 сообщает пользователю размер группы,
     * к которой прикреплена область связи,
     * к которой прикреплен описатель (коммуникатор) MPI_COMM_WORLD,
     * т.е. число процессов в приложении!!
     */
   if( rank==0 )
      printf("Total processes count = %d\n", size );

    /* Каждая задача выводит пользователю свой номер */
   printf("Hello! My rank in MPI_COMM_WORLD = %d\n", rank );

    /* Точка синхронизации, затем задача 0 печатает
     * аргументы командной строки. В командной строке
     * могут быть параметры, добавляемые загрузчиком MPIRUN.
     */
   MPI_Barrier( MPI_COMM_WORLD );
if( rank == 0 )
      for( puts ("Command line of process 0:"),  i=0; i<argc;i++)
         printf( "%d: \"%s\"\n", i, argv[i] );
/* Все задачи завершают выполнение */
   MPI_Finalize();
   return 0;
}
```

9.2.5 Функции пересылки данных.

Хотя с теоретической точки зрения ветвям для организации обмена данными достаточно всего двух операций (прием и передача), на практике все обстоит гораздо сложнее. Одними только коммуникациями "точка-точка" (т.е. такими, в которых ровно один передающий процесс и ровно один принимающий) занимается порядка 40 функций. Пользуясь ими, программист имеет возможность выбрать:

- способ зацепления процессов - в случае неодновременного вызова двумя процессами парных функций приема и передачи могут быть произведены:
- автоматический выбор одного из трех нижеприведенных вариантов;
    - буферизация на передающей стороне - функция передачи заводит временный буфер, копирует в него сообщение и возвращает управление вызвавшему процессу. Содержимое буфера будет передано в фоновом режиме;
    - ожидание на приемной стороне, завершение с кодом ошибки на передающей стороне;

  - ожидание на передающей стороне, завершение с кодом ошибки на приемной стороне.
- способ взаимодействия коммуникационного модуля MPI с вызывающим процессом:
  - блокирующий - управление вызывающему процессу возвращается только после того, как данные приняты или переданы (или скопированы во временный буфер);
  - неблокирующий - управление возвращается немедленно (т.е. процесс блокируется до завершения операции), и фактическая приемопередача происходит в фоне. Функция неблокирующего приема имеет дополнительный параметр типа "квитанция". Процесс не имеет права производить какие-либо действия с буфером сообщения, пока квитанция не будет "погашена";
- персистентный - в отдельные функции выделены:
  - создание "канала" для приема/передачи сообщения,
  - инициация приема/передачи,
  - закрытие канала.

Такой способ эффективен, к примеру, если приемопередача происходит внутри цикла, а создание/закрытие канала вынесены за его границы.

Две простейшие (но и самые медленные) функции - MPI_Recv и MPI_Send - выполняют блокирующую приемопередачу с автоматическим выбором зацепления (кстати сказать, все функции приема совместимы со всеми функциями передачи). Таким образом, MPI - весьма разветвленный инструментарий. То, что в конкурирующих пакетах типа PVM реализовано одним-единственным способом, в MPI может быть сделано несколькими, про которые говорится: способ А прост в использовании, но не очень эффективен; способ Б сложнее, но эффективнее; а способ В сложнее и эффективнее при определенных условиях. Замечание о разветвленности относится и к коллективным коммуникациям (при которых получателей и/или отправителей несколько): в MPI эта категория представлена 9 функциями 5 типов:
broadcast: один-всем,
scatter: один-каждому,
gather: каждый-одному,
allgather: все-каждому,
alltoall: каждый-каждому.

На первый взгляд может показаться, что программисту легче будет в случае необходимости самому написать такую функцию, но при этом он, скорее всего, будет использовать функции типа MPI_Send и MPI_Recv, в то время как имеющиеся в MPI функции оптимизированы - не пользуясь функциями "точка-точка", они напрямую (на что, согласно идеологии MPI, программа пользователя права не имеет) обращаются к разделяемой памяти и семафорам и к TCP/IP при работе в сети. А если такая архитектурно-зависимая оптимизация невозможна, используется оптимизация архитектурно-независимая: передача производится не напрямую от одного ко всем (время передачи линейно зависит от количества ветвей-получателей), а по двоичному дереву (время передачи логарифмически зависит от количества). Как следствие, скорость работы повышается.

9.2.6 Связь "точка-точка". Простейший набор. Пример.

Это самый простой тип связи между задачами: одна ветвь вызывает функцию передачи данных, а другая - функцию приема. В MPI это выглядит, например, так:
Задача 1 передает:

```
int buf[10];
MPI_Send( buf, 5, MPI_INT, 1, 0, MPI_COMM_WORLD );
```

Задача 2 принимает:

```
int buf[10];
MPI_Status status;
MPI_Recv( buf, 10, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
```

Аргументы функций:

Адрес буфера, из которого в задаче 1 берутся, а в задаче 2 помещаются данные. Наборы данных у каждой задачи свои, поэтому, например, используя одно и то же имя массива в нескольких задачах, указываете не одну и ту же область памяти, а разные, никак друг с другом не связанные.

Размер буфера. Задается **не в байтах**, а в количестве ячеек. Для MPI_Send указывает, сколько ячеек требуется передать (в примере передаются 5 чисел). В MPI_Recv означает максимальную емкость приемного буфера. Если фактическая длина пришедшего сообщения меньше - последние ячейки буфера останутся нетронутыми, если больше - произойдет ошибка времени выполнения.

Тип ячейки буфера. MPI_Send и MPI_Recv оперируют массивами однотипных данных. Для описания базовых типов Си в MPI определены константы MPI_INT, MPI_CHAR, MPI_DOUBLE и так далее, имеющие тип MPI_Datatype. Их названия образуются префиксом "MPI_" и именем соответствующего типа (int, char, double, ...), записанным заглавными буквами. Пользователь может "регистрировать" в MPI свои собственные типы данных, например, структуры, после чего MPI сможет обрабатывать их наравне с базовыми.

Номер задачи, с которой происходит обмен данными. Все задачи внутри созданной MPI группы автоматически нумеруются от 0 до (размер группы-1). В примере задача 0 передает задаче 1, задача 1 принимает от задачи 0.

Идентификатор сообщения. Это целое число от 0 до 32767, которое пользователь выбирает сам. Оно служит той же цели, что и, например, расширение файла - задача-приемник:
по идентификатору определяет смысл принятой информации ;
сообщения, пришедшие в неизвестном порядке, может извлекать из общего входного потока в нужном алгоритму порядке. Хорошим тоном является обозначение идентификаторов символьными именами посредством операторов "#define" или "const int".

Описатель области связи (коммуникатор). Обязан быть одинаковым для MPI_Send и MPI_Recv.

Статус завершения приема. Содержит информацию о принятом сообщении: его идентификатор, номер задачи-передатчика, код завершения и количество фактически пришедших данных.

9.2.7 Коллективные функции.

Под термином "коллективные" в MPI подразумеваются три группы функций:

- точки синхронизации, или барьеры;
- функции коллективного обмена данными (о них уже упоминалось выше);
- функции поддержки распределенных операций.

Коллективная функция одним из аргументов получает описатель области связи (коммуникатор). Вызов коллективной функции является корректным, только если произведен из всех процессов-абонентов соответствующей области связи, и именно с этим коммуникатором в качестве аргумента (хотя для одной области связи может иметься несколько коммуникаторов, подставлять их вместо друг друга нельзя). В этом и заключается коллективность: либо функция вызывается всем коллективом процессов, либо никем; третьего не дано.

Как поступить, если требуется ограничить область действия для коллективной функции только частью присоединенных к коммуникатору задач, или наоборот - расширить область

действия? Нужно создавать временную группу/область связи/коммуникатор на базе существующих.

Основные особенности и отличия от коммуникаций типа "точка-точка":

- на прием и/или передачу работают одновременно все задачи-абоненты указываемого коммуникатора;
- коллективная функция выполняет одновременно и прием, и передачу; она имеет большое количество параметров, часть которых нужна для приема, а часть для передачи; в разных задачах та или иная часть игнорируется;
- как правило, значения ВСЕХ параметров (за исключением адресов буферов) должны быть идентичными во всех задачах;
- MPI назначает идентификатор для сообщений автоматически; кроме того, сообщения передаются не по указываемому коммуникатору, а по временному коммуникатору-дупликату; тем самым потоки данных коллективных функций надежно изолируются друг от друга и от потоков, созданных функциями "точка-точка".

MPI_Bcast рассылает содержимое буфера из задачи, имеющей в указанной области связи номер **root**, во все остальные:

```
MPI_Bcast( buf, count, dataType, rootRank, communicator );
```

MPI_Gather ("совок") собирает в приемный буфер задачи root передающие буфера остальных задач.

Векторный вариант "совка" - **MPI_Gatherv** - позволяет задавать разное количество отправляемых данных в разных задачах-отправителях. Соответственно, на приемной стороне задается массив позиций в приемном буфере, по которым следует размещать поступающие данные, и максимальные длины порций данных от всех задач. Оба массива содержат позиции/длины **не** в байтах, а в количестве ячеек типа recvCount.

MPI_Scatter ("разбрызгиватель") : выполняет обратную "совку" операцию - части передающего буфера из задачи root распределяются по приемным буферам всех задач

И ее векторный вариант - **MPI_Scatterv**, рассылающая части неодинаковой длины в приемные буфера неодинаковой длины.

MPI_Allgather аналогична MPI_Gather, но прием осуществляется не в одной задаче, а во ВСЕХ: каждая имеет специфическое содержимое в передающем буфере, и все получают одинаковое содержимое в буфере приемном. Как и в MPI_Gather, приемный буфер последовательно заполняется данными изо всех передающих. Вариант с неодинаковым количеством данных называется **MPI_Allgatherv**.

MPI_Alltoall : каждый процесс нарезает передающий буфер на куски и рассылает куски остальным процессам; каждый процесс получает куски от всех остальных и поочередно размещает их приемном буфере. Это "совок" и "разбрызгиватель" в одном флаконе. Векторный вариант называется **MPI_Alltoallv**.

Коллективные функции несовместимы с "точка-точка": недопустимым, например, является вызов в одной из принимающих широковещательное сообщение задач MPI_Recv вместо MPI_Bcast.

9.2.8 Точки синхронизации, или барьеры.

Этим занимается всего одна функция:

```
int MPI_Barrier( MPI_Comm comm );
```

MPI_Barrier останавливает выполнение вызвавшей ее задачи до тех пор, пока не будет вызвана изо всех остальных задач, подсоединенных к указываемому коммуникатору. Гарантирует, что к выполнению следующей за MPI_Barrier инструкции каждая задача приступит одновременно с остальными.

Это единственная в MPI функция, вызовами которой гарантированно синхронизируется во времени выполнение различных ветвей! Некоторые другие коллективные функции в зависимости от реализации могут обладать, а могут и не обладать свойством одновременно возвращать управление всем ветвям; но для них это свойство является побочным и необязательным - если Вам нужна синхронность, используйте только MPI_Barrier.
Когда может потребоваться синхронизация? Например, синхронизация используется перед аварийным завершением.
Это утверждение непроверено, но: алгоритмическое необходимости в барьерах, как представляется, нет. Параллельный алгоритм для своего описания требует по сравнению с алгоритмом классическим всего лишь двух дополнительных операций - приема и передачи из ветви в ветвь. Точки синхронизации несут чисто технологическую нагрузку вроде той, что описана в предыдущем абзаце.
Иногда случается, что ошибочно работающая программа перестает врать, если ее исходный текст хорошенько нашпиговать барьерами. Однако программа начнет работать медленнее, например:

```
Без барьеров:           0    xxxx....xxxxxxxxxxxxxxxxxxxx
                      1    xxxxxxxxxxxx....xxxxxxxxxxxx
                                 2    xxxxxxxxxxxxxxxxxxxxxx....xx
С барьерами:           0    xxxx....xx(xxxxxxxx(||||xxxxxxxx(||xx
                                 1    xxxxxx(||||x....xxxxxxx(xxxxxxxx(||xx
                                 2    xxxxxx(||||xxxxxxxx(||||..xxxxxxxx(xx
                                       ---------------------------- > Время

Обозначения:
    x   нормальное выполнение
    .   ветвь простаивает - процессорное время отдано под другие цели
    (   вызван MPI_Barrier
    |   MPI_Barrier ждет своего вызова в остальных ветвях
```

Так что "задавить" ошибку барьерами хорошо только в качестве временного решения на период отладки.

9.2.9 Распределенные операции.

Идея проста: в каждой задаче имеется массив. Над нулевыми ячейками всех массивов производится некоторая операция (сложение/произведение/ поиск минимума/максимума и т.д.), над первыми ячейками производится такая же операция и т.д. Четыре функции предназначены для вызова этих операций и отличаются способом размещения результата в задачах.
MPI_Reduce : массив с результатами размещается в задаче с номером **root**:

```
int vector[16];
int resultVector[16];
MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
for( i=0; i<16; i++ )
    vector[i] = myRank*100 + i;
         MPI_Reduce(
    vector,             /* каждая задача в коммуникаторе предоставляет вектор */
    resultVector,   /* задача номер 'root' собирает данные сюда */
     16,                        /* количество ячеек в исходном и результирующем
массивах */
    MPI_INT,       /* и тип ячеек */
    MPI_SUM,    /* описатель операции: поэлементное сложение векторов */
    0,                   /* номер задачи, собирающей результаты в 'resultVector'
*/
    MPI_COMM_WORLD  /* описатель области связи */
);
if( myRank==0 )
    /* печатаем resultVector, равный сумме векторов */
```

Предопределенных описателей операций в MPI насчитывается 12:
1. **MPI_MAX** и **MPI_MIN** ищут поэлементные максимум и минимум;
2. **MPI_SUM** вычисляет сумму векторов;
3. **MPI_PROD** вычисляет поэлементное произведение векторов;
4. **MPI_LAND**, **MPI_BAND**, **MPI_LOR**, **MPI_BOR**, **MPI_LXOR**, **MPI_BXOR** - логические и двоичные операции И, ИЛИ, исключающее ИЛИ;
5. **MPI_MAXLOC**, **MPI_MINLOC** - поиск индексированного минимума/максимума.

Количество поддерживаемых операциями типов для ячеек векторов строго ограничено вышеперечисленными. Никакие другие встроенные или пользовательские описатели типов использоваться не могут! Все операции являются ассоциативными ("(a+b)+c = a+(b+c)") и коммутативными ("a+b = b+a").
MPI_Allreduce : результат рассылается всем задачам, параметр 'root' убран.
MPI_Reduce_scatter : каждая задача получает не весь массив-результат, а его часть. Длины этих частей находятся в массиве-третьем параметре функции. Размер исходных массивов во всех задачах одинаков и равен сумме длин результирующих массивов.
MPI_Scan : аналогична функции MPI_Allreduce в том отношении, что каждая задача получает результрующий массив. Главное отличие: здесь содержимое массива-результата в задаче **i** является результатом выполнение операции над массивами из задач с номерами от **0** до **i** включительно.
Помимо встроенных, пользователь может вводить свои собственные операции.

9.2.10 Создание коммуникаторов и групп.

Копирование. Самый простой способ создания коммуникатора - скопировать "один-в-один" уже имеющийся:

```
MPI_Comm tempComm;
MPI_Comm_dup( MPI_COMM_WORLD, &tempComm );
 /* ... передаем данные через tempComm ... */
MPI_Comm_free( &tempComm );
```

Новая группа при этом не создается - набор задач остается прежним. Новый коммуникатор наследует все свойства копируемого.

Расщепление. Соответствующая коммуникатору группа расщепляется на непересекающиеся подгруппы, для каждой из которых заводится свой коммуникатор.

```
MPI_Comm_split(
    existingComm,                          /* существующий описатель, например
MPI_COMM_WORLD */
    indexOfNewSubComm,    /* номер подгруппы, куда надо поместить ветвь */
    rankInNewSubComm,      /* желательный номер в новой подгруппе */
    &newSubComm );            /* описатель области связи новой подгруппы */
```

Эта функция имеет одинаковый первый параметр во всех ветвях, но разные второй и третий - и в зависимости от них разные ветви определяются в разные подгруппы; возвращаемый в четвертом параметре описатель будет принимать в разных ветвях разные значения (всего столько разных значений, сколько создано подгрупп). Если indexOfNewSubComm равен **MPI_UNDEFINED**, то в newSubComm вернется MPI_COMM_NULL, то есть ветвь не будет включена ни в какую из созданных групп.

Создание через группы. В предыдущих двух случаях коммуникатор создается от существующего коммуникатора напрямую, без явного создания группы: группа либо та же самая, либо создается автоматически. Самый же общий способ таков:

функцией **MPI_Comm_group** определяется группа, на которую указывает соответствующий коммуникатор;

на базе существующих групп функциями семейства **MPI_Group_xxx** создаются новые группы с нужным набором ветвей;

для итоговой группы функцией **MPI_Comm_create** создается коммуникатор. Она должна быть вызвана во **всех** ветвях-абонентах коммуникатора, передаваемого первым параметром;

все описатели созданных групп очищаются вызовами функции **MPI_Group_free**.

Такой механизм позволяет не только расщеплять группы подобно MPI_Comm_split, но и объединять их. Всего в MPI определено 7 разных функций конструирования групп.

9.2.11 MPI и типы данных.

О типах передаваемых данных MPI должен знать постольку-поскольку при работе в сетях на разных машинах данные могут иметь разную разрядность (например, тип *int* - 4 или 8 байт), ориентацию (младший байт располагается в ОЗУ первым на процессорах Intel, последним - на всех остальных), и представление (это, в первую очередь, относится к размерам мантиссы и экспоненты для вещественных чисел). Поэтому все функции приемопередачи в MPI оперируют не количеством передаваемых байт, а количеством ячеек, тип которых задается параметром функции, следующим за количеством: MPI_INTEGER, MPI_REAL и т.д. Это переменные типа MPI_Datatype (тип "описатель типов", каждая его переменная описывает для MPI один тип). Они имеются для каждого базового типа, имеющегося в используемом языке программирования.

Однако, пользуясь базовыми описателями, можно передавать либо массивы, либо одиночные ячейки (как частный случай массива). А как передавать данные агрегатных типов, например, структуры? В MPI имеется механизм конструирования пользовательских описателей на базе уже имеющихся (как пользовательских, так и встроенных).

Более того, разработчики MPI создали механизм конструирования новых типов даже более универсальный, чем имеющийся в языке программирования. Действительно, во всех языках программирования ячейки внутри агрегатного типа (массива или структуры):

не налезают друг на друга,

не располагаются с разрывами (выравнивание полей в структурах не в счет).

В MPI сняты оба этих ограничения. Это позволяет весьма причудливо "вырезать", в частности, фрагменты матриц для передачи, и размещать принимаемые данные между собственных.

Выигрыш от использования механизма конструирования типов очевиден - лучше один раз вызвать функцию приемопередачи со сложным шаблоном, чем двадцать раз - с простыми.

9.2.12 Зачем MPI знать тип передаваемых данных?

Действительно, зачем? Стандартные функции пересылки данных прекрасно обходятся без подобной информации - им требуется знать только размер в байтах. Вместо одного такого аргумента функции MPI получают два: количество элементов некоторого типа и символический описатель указанного типа (MPI_INT, и т.д.). Причин тому несколько:

- Пользователю MPI позволяет описывать свои собственные типы данных, которые располагаются в памяти не непрерывно, а с разрывами, или наоборот, с "налезаниями" друг на друга. Переменная такого типа характеризуется не только размером, и эти характеристики MPI хранит в описателе типа.
- Приложение MPI может работать на гетерогенном вычислительном комплексе (коллективе ЭВМ с разной архитектурой). Одни и те же типы данных на разных машинах могут иметь разное представление, например: на плавающую арифметику существует 3 разных стандарта (IEEE,IBM,Cray); тип char в терминальных приложениях Windows представлен альтернативной кодировкой ГОСТ, а в Юниксе - кодировкой KOI-8r ; ориентация байтов в многобайтовых числах на ЭВМ с процессорами Intel отличается от общепринятой (у Intel - младший байт занимает младший адрес, у всех остальных - наоборот). Если приложение работает в

гетерогенной сети, через сеть задачи обмениваются данными в формате XDR (eXternal Data Representation), принятом в Internet. Перед отправкой и после приема данных задача конвертирует их в/из формата XDR. Естественно, при этом MPI должен знать не просто количество передаваемых байт, но и тип содержимого.

- Обязательным требованием к MPI была поддержка языка Фортран в силу его инерционной популярности. Фортрановский тип CHARACTER требует особого обращения, поскольку переменная такого типа содержит не собственно текст, а адрес текста и его длину. Функция MPI, получив адрес переменной, должна извлечь из нее адрес текста и копировать сам текст. Это и произойдет, если в поле аргумента-описателя типа стоит MPI_CHARACTER. Ошибка в указании типа приведет: при отправке - к копированию служебных данных вместо текста, при приеме - к записи текста на место служебных данных. И то, и другое приводит к ошибкам времени выполнения.
- Такие часто используемые в Си типы данных, как структуры, могут содержать в себе некоторое пустое пространство, чтобы все поля в переменной такого типа размещались по адресам, кратным некоторому четному числу (часто 2, 4 или 8) - это ускоряет обращение к ним. Причины тому чисто аппаратные. Выравнивание данных настраивается ключами компилятора. Разные задачи одного и того же приложения, выполняющиеся на одной и той же машине (даже на одном и том же процессоре), могут быть построены с разным выравниванием, и типы с одинаковым текстовым описанием будут иметь разное двоичное представление. MPI будет вынужден позаботиться о правильном преобразовании. Например, переменные такого типа могут занимать 9 или 16 байт:

```
typedef struct {
    char    c;
    double  d;
} CharDouble;
```

9.2.13 Использование MPI.

MPI сам по себе является средством:

- сложным: спецификация на MPI-1 содержит 300 страниц, на MPI-2 - еще 500 (причем это *только отличия и добавления* к MPI-1), и программисту для эффективной работы так или иначе придется с ними ознакомиться, помнить о наличии нескольких сотен функций, и о тонкостях их применения;
- специализированным: это система связи.

Можно сказать, что сложность (т.е. многочисленность функций и обилие аргументов у большинства из них) является ценой за компромисс между эффективностью и универсальностью. С одной стороны, на SMP-машине должны существовать способы получить *почти* столь же высокую скорость при обмене данными между ветвями, как и при традиционном программировании через разделяемую память и семафоры. С другой стороны, все функции должны работать на любой платформе. Таким образом, программист заинтересован в инструментах, которые облегчали бы:

проведение декомпозиции,
запись ее в терминах MPI.

В данном случае это средства, генерирующие на базе входных данных текст программы на стандартном Си или Фортране, обладающей явным параллелизмом, выраженным в терминах MPI; содержащий вызовы MPI-процедур, наиболее эффективные в окружающем контексте. Такие средства делают написание программы не только легче, но и надежнее. Назовем

некоторые перспективные типы такого инструментария, который лишал бы программиста необходимости вообще помнить о присутствии MPI.

Средства автоматической декомпозиции. Идеалом является такое оптимизирующее средство, которое на входе получает исходный текст некоего последовательного алгоритма, написанный на обычном языке программирования, и выдает на выходе исходный текст этого же алгоритма на этом же языке, но уже в распараллеленном на ветви виде, с вызовами MPI. Что ж, такие средства созданы, но никто не торопится раздавать их бесплатно. Кроме того, вызывает сомнение их эффективность.

Языки программирования. Это наиболее популярные на сегодняшний день средства полуавтоматической декомпозиции. В синтаксис универсального языка программирования (Си или Фортрана) вводятся дополнения для записи параллельных конструкций кода и данных. Препроцессор переводит текст в текст на стандартном языке с вызовами MPI. Примеры таких систем: mpC (massively parallel C) и HPF (High Performance Fortran).

Оптимизированные библиотеки для стандартных языков. В этом случае оптимизация вообще может быть скрыта от проблемного программиста. Чем больший объем работы внутри программы отводится подпрограммам такой библиотеки, тем большим будет итоговый выигрыш в скорости ее (программы) работы. Собственно же программа пишется на обычном языке программирования безо всяких упоминаний об MPI, и строится стандартным компилятором. От программиста потребуется лишь указать для компоновки имя библиотечного файла MPI, и запускать полученный в итоге исполняемый код не непосредственно, а через MPI-загрузчик. Популярные библиотеки обработки матриц, такие как Linpack, Lapack и ScaLapack, уже переписаны под MPI.

Средства визуального проектирования. Действительно, почему бы не расположить на экране несколько окон с исходным текстом ветвей, и пусть пользователь легким движением мыши протягивает стрелки от точек передачи к точкам приема - а визуальный построитель генерирует полный исходный текст?

Отладчики. Об отладчиках пока можно сказать только то, что они нужны. Должна быть возможность одновременной трассировки/просмотра нескольких параллельно работающих ветвей - что-либо более конкретное пока сказать трудно.

9.2.14 MPI-1 и MPI-2.

В функциональности MPI есть пробелы, которые устранены в следующем проекте, MPI-2. Вкратце перечислим наиболее важные нововведения:

- Взаимодействие между приложениями. Поддержка механизма "клиент-сервер".
- Динамическое порождение ветвей.
- Для работы с файлами создан архитектурно-независимый интерфейс.
- Сделан шаг в сторону SMP-архитектуры. Теперь разделяемая память может быть не только каналом связи между ветвями, но и местом совместного хранения данных.

9.2.15 Пример.

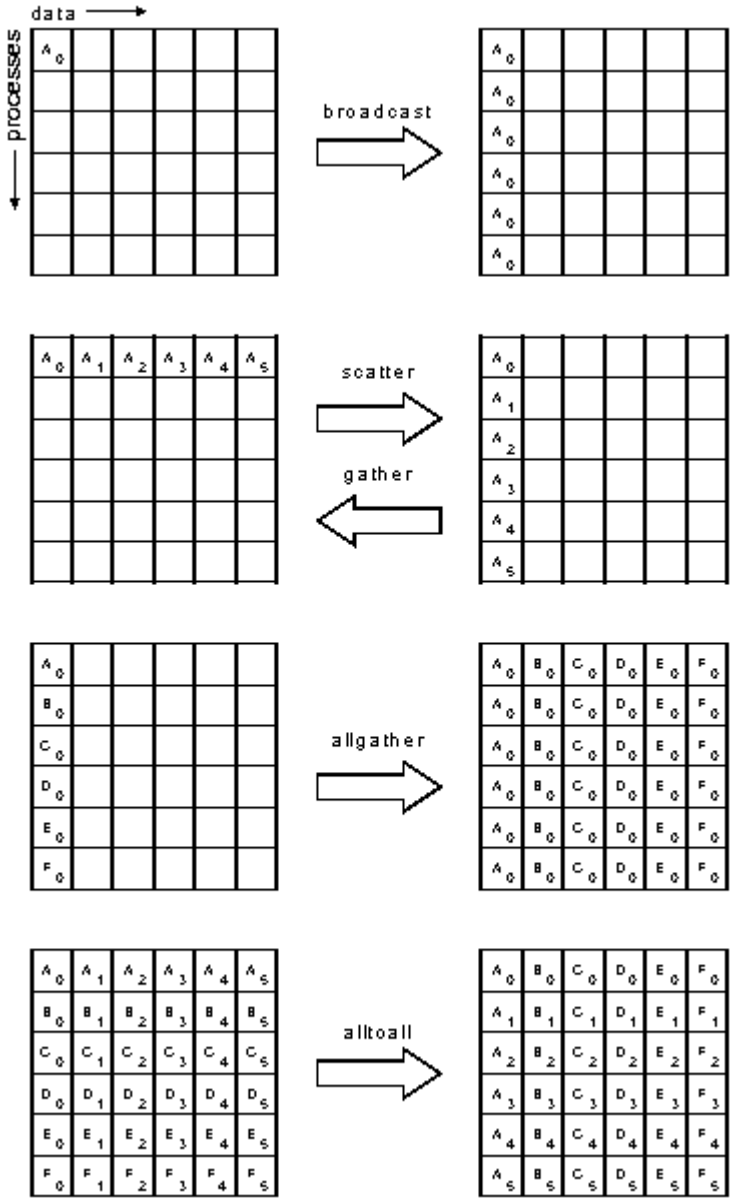
```
/*
 *  Простейшая приемопередача:
 *     MPI_Send, MPI_Recv
 *  Завершение по ошибке:
 *     MPI_Abort
 */
```

```
#include <mpi.h>
#include <stdio.h>

  /* Идентификаторы сообщений */
#define tagFloatData  1
#define tagDoubleData 2

  /* Этот макрос введен для удобства, */
  /* он позволяет указывать длину массива в количестве ячеек */
#define ELEMS(x)  ( sizeof(x) / sizeof(x[0]) )

int main( int argc, char **argv )
{
    int size, rank, count;
    float floatData[10];
    double doubleData[20];
    MPI_Status status;
      /* Инициализируем библиотеку */
    MPI_Init( &argc, &argv );
      /* Узнаем количество задач в запущенном приложении */
    MPI_Comm_size( MPI_COMM_WORLD, &size );
      /* ... и свой собственный номер: от 0 до (size-1) */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
      /* пользователь должен запустить ровно две задачи, иначе ошибка */
    if( size != 2 ) {
          /* задача с номером 0 сообщает пользователю об ошибке */
        if( rank==0 )
            printf("Error: two processes required instead of %d, abort\n",
                size );
          /* Все задачи-абоненты области связи MPI_COMM_WORLD
           * будут стоять, пока задача 0 не выведет сообщение.
           */
        MPI_Barrier( MPI_COMM_WORLD );
          /* Без точки синхронизации может оказаться, что одна из задач
           * вызовет MPI_Abort раньше, чем успеет отработать printf()
           * в задаче 0, MPI_Abort немедленно принудительно завершит
           * все задачи и сообщение выведено не будет
           */
          /* все задачи аварийно завершают работу */
        MPI_Abort(
            MPI_COMM_WORLD,  /* Описатель области связи, на которую */
                             /*    распространяется действие ошибки */
            MPI_ERR_OTHER ); /* Целочисленный код ошибки */
        return -1;
    }
    if( rank==0 ) {
      /* Задача 0 что-то такое передает задаче 1 */
        MPI_Send(
            floatData,        /* 1) адрес передаваемого массива */
            5,       /* 2) сколько: 5 ячеек, т.е. floatData[0]..floatData[4] */
            MPI_FLOAT,        /* 3) тип ячеек */
            1,                /* 4) кому: задаче 1 */
            tagFloatData,     /* 5) идентификатор сообщения */
            MPI_COMM_WORLD ); /* 6) описатель области связи, через которую */
                              /*    происходит передача */
          /* и еще одна передача: данные другого типа */
        MPI_Send( doubleData, 6, MPI_DOUBLE, 1, tagDoubleData, MPI_COMM_WORLD );
    } else {
      /* Задача 1 что-то такое принимает от задачи 0 */
          /* дожидаемся сообщения и помещаем пришедшие данные в буфер */
        MPI_Recv(
            doubleData,     /* 1) адрес массива, куда складывать принятое */
            ELEMS( doubleData ), /* 2) фактическая длина приемного */
```

```
                                /*    массива в числе ячеек */
            MPI_DOUBLE,     /* 3) сообщаем MPI, что пришедшее сообщение */
                            /*    состоит из чисел типа 'double' */
            0,              /* 4) от кого: от задачи 0 */
            tagDoubleData,  /* 5) ожидаем сообщение с таким идентификатором */
            MPI_COMM_WORLD, /* 6) описатель области связи, через которую */
                            /*    ожидается приход сообщения */
            &status );      /* 7) сюда будет записан статус завершения приема */

          /* Вычисляем фактически принятое количество данных */
        MPI_Get_count(
            &status,        /* статус завершения */
            MPI_DOUBLE,     /* сообщаем MPI, что пришедшее сообщение */
                            /*    состоит из чисел типа 'double' */
            &count );       /* сюда будет записан результат */

          /* Выводим фактическую длину принятого на экран */
        printf("Received %d elems\n", count );

          /* Аналогично принимаем сообщение с данными типа float
           * Обратите внимание: задача-приемник имеет возможность
           * принимать сообщения не в том порядке, в котором они
           * отправлялись, если эти сообщения имеют разные идентификаторы
           */
        MPI_Recv( floatData, ELEMS( floatData ), MPI_FLOAT,
            0, tagFloatData, MPI_COMM_WORLD, &status );
        MPI_Get_count( &status, MPI_FLOAT, &count );
    }
      /* Обе задачи завершают выполнение */
    MPI_Finalize();
    return 0;
}
```

Приложение.

Коллективные функции:


data
processes
broadcast
scatter
gather
allgather
alltoall

Общие функции

int MPI_Init(int* argc, char* argv)**
int MPI_Finalize(void)

int MPI_Comm_size(MPI_Comm comm, int* size)
int MPI_Comm_rank(MPI_comm comm, int* rank)

double MPI_Wtime(void)

int MPI_Barrier(MPI_Comm comm)

9.2.16 Прием-передача - базовые

int MPI_{I}Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm{, MPI_Request *request})
buf - адрес начала буфера посылки сообщения
count - число передаваемых элементов в сообщении
datatype - тип передаваемых элементов
dest - номер процесса-получателя
msgtag - идентификатор сообщения
comm - идентификатор группы

int MPI_{I}Recv(void* buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_comm comm, MPI_Status *status{, MPI_Request *request})
buf - адрес начала буфера приема сообщения
count - максимальное число элементов в принимаемом сообщении
datatype - тип элементов принимаемого сообщения
source - номер процесса-отправителя
msgtag - идентификатор принимаемого сообщения
comm - идентификатор группы
status - параметры принятого сообщения

int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status)
source - номер процесса-отправителя или *MPI_ANY_SOURCE*
msgtag - идентификатор ожидаемого сообщения или *MPI_ANY_TAG*
comm - идентификатор группы
status - параметры обнаруженного сообщения

int MPI_Get_Count(MPI_Status *status, MPI_Datatype datatype, int *count)
status - параметры принятого сообщения
datatype - тип элементов принятого сообщения
count - число элементов сообщения

Пример:

```
MPI_Probe( MPI_ANY_SOURCE, tagMessageInt, MPI_COMM_WORLD, &status );
MPI_Get_count( &status, MPI_INT, &bufElems );
buf = malloc( sizeof(int) * bufElems );
MPI_Recv( buf, bufElems, MPI_INT, ...
```

Асинхронные прием-передача: действия с квитанциями

int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_WaitAll(int count, MPI_Request *requests, MPI_Status *statuses)
int MPI_WaitAny(int count, MPI_Request *requests, int *index, MPI_Status *status)
int MPI_WaitSome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_TestAll(int count, MPI_Request *requests, int *flag, MPI_STatus *statuses)
int MPI_TestAny(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status)
int MPI_TestSome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

Объединение запросов

int MPI_Send_Init(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)

int MPI_Recv_Init(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Request *request)

MPI_Start_All(int count, MPI_Request *requests)