Московский государственный университет
им. Ломоносова
Факультет вычислительной математики и кибернетики

Конспект лекций
По курсу

«Операционные системы»

(Лектор Машечкин И.В.)

Выполнила студентка 208 группы
Лукьяница Василиса Андреевна

Москва, 2003

Лекция 1. Операционные системы.

План

1) Введение:
   - историч. развитие ВС, определение понятия ВС, появление и развитие программного обеспечения;
   - основные компоненты совр. Компьютеров;
   - обзор свойств и характеристик совр. Компьютеров;
2) Методы и принципы ОС:
   - процессы;
   - файловые системы;
   - планирование в ОС;
   - организация управления внешними устройствами;
   - управление оперативной памятью, сетевое взаимодействие.

Экскурс в историю.

Первое поколение компьютеров.
Компьютеры 1ого поколения относятся к концу 40х гг. считается, что они возникли в результате развития ядерного оружия.
Основная модель – ENIAC, устройство которой было основано на лампах, за счет чего компьютер был достаточно большого размера, имел маленькую производительность и работал только в персональном режиме.

Основные трудности в работе с такой машиной:

- программисту не просто было необходимо знать все системные особенности компьютера, но и вводить данные со специального пульта в двоичном (машинном) коде;
- в случае аварийной ситуации компьютер останавливал работу и необходимо было искать ошибку в двоичном коде;
- трудно было изменять программу, т.к. использовалась безусловная ядресация
- возникали проблемы в работе с внешними устройствами.

На этом же этапе зародились первые сервисные программы с мнемоническими обозначениями => assembler => трансляторы с asm в машинный код => программы управления внешними устройствами.

Компьютеры второго поколения.
Компьютеры второго поколения датируются концом 50х – второй половиной 60х гг. Они основаны уже на полупроводниковых приборах – диодах и транзисторах, поэтому их размер меньше, уменьшилось потребление энергии, но увеличилась скорость работы.

Этот этап х-ся появлением и развитием ПО:

- внешние устройства этих машин (магнитные ленты) очень медленные => появление мультипрограммных систем;
- появление языков управления заданиями, в которых декларировались ресурсы, необходимые для программы, такие как максимальное время выполнения, максимальный необходимый объем оперативной памяти и т.п.
- необходимо было знать все интерфейсы внешних устройств => появление файловой системы – возможности именовать данные и иметь доступ к ним;
- развитие внешних устройств => масса управляющих программ очень велика => виртуальные устройства (процесс обобщения св-в конкретных аппаратных устройств и объединение нескольких групп св-в, напр., файловая система).

Компьютеры третьего поколения.
Конец 60х – 70х гг, основаны на интегральных схемах алой интеграции. Массовое внедрение выч. Технологий в управление производством и активное развитие периферии.

Характерно:

- унификация узлов и устройств для совместимости различных моделей;
- появление «семейств» компьютеров, для преемственности программ компьютерами различных моделей снизу вверх;
- большое развитие получили ОС, напр., UNIX =>появление драйверов.

Компьютеры четвертого и последующих поколений.
Осн. х-ка – использование интегральных схем большой и сверхбольшой интеграции.

- потребность создания максимально «дружественных» систем => «дружественные пользовательские интерфейсы; возрождение понятия персональный компьютер => массовое распределение ПК по всем нишам социума;
- толчок к развитию сетевых технологий: первоначально корпоративные сети слишком закрытые, но кол-во информации требовало унификации сетей;
- проблема обеспечения безопасности хранения и передачи данных.

Основы архитектуры вычислительных систем.

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

Элементарными примерами ВС могут служить игровые автоматы и мобильные телефоны.

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

Взаимодействие уровней происходит ввиду непосредственных уровневых интерфейсов или какого либо косвенного влияния (напр. Если канал рассчитан на 10 человек, а используется 100).

Аппаратные средства ЭВМ

С позиции уровней выше, т.н. физические ресурсы, каждому из которых соотв. определенные характеристики и аппаратные компоненты. Физ.ресурсы: процессор, Оперативная память, наличие\отсутствие внешних устройств.

Характеристики каждого ресурся (в идеале):

- правила программного использования;
- производительность или емкость;
- степень занятости или используемости.

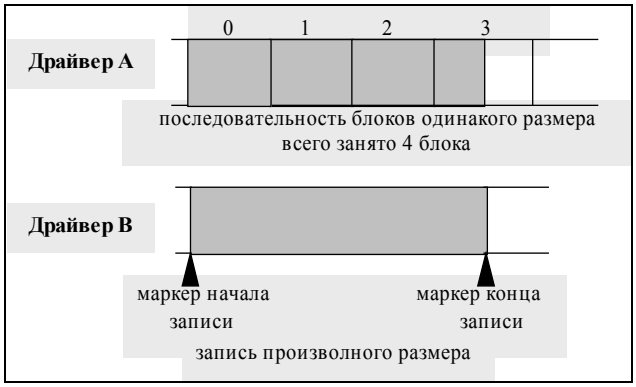
Но не существует единого правила формирования этих характеристик, в т.ч. одно и то же устройство может обладать различными характеристиками в зависимости от назначения.

Уровень аппаратных средств ЭВМ – система команд ЭВМ и программно управляемые компоненты ЭВМ.

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.

Пример различных драйверов для магнитной ленты.



программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством.
Пользователю доступны:

- системы команд компьютера;
- аппаратные средства;
- программные интерфейсы доступа через соответствующие драйверы.

Возникающие проблемы:

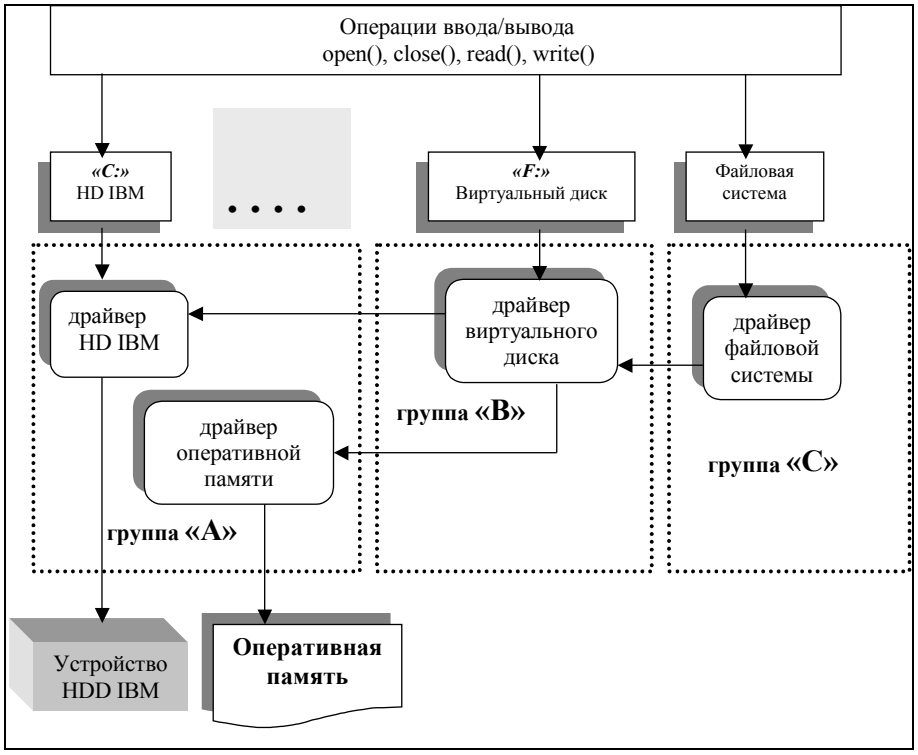
- должна использоваться модификация для перехода от одного устройства к другому;
- необходимо организовывать одновременную работу большого кол-ва устройств.

Управление логическими/виртуальными ресурсами

Основа: обобщение свойств и интерфейсов физ. устройств и унификация интерфейсов.

Логическое/виртуальное устройство (ресурс) – это устройство/ресурс, некоторые эксплутационные характеристики которого (возможно все) реализованы программным образом.

Пример.



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.

Ресурсы вычислительной системы - совокупность всех физических и виртуальных ресурсов.

Одна из характеристик ресурсов вычислительной системы их **конечность**, следовательно возникает конкуренция за обладание ресурсом между его программными потребителями.

Операционная система - это комплекс программ, обеспечивающий управление ресурсами вычислительной системы.

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

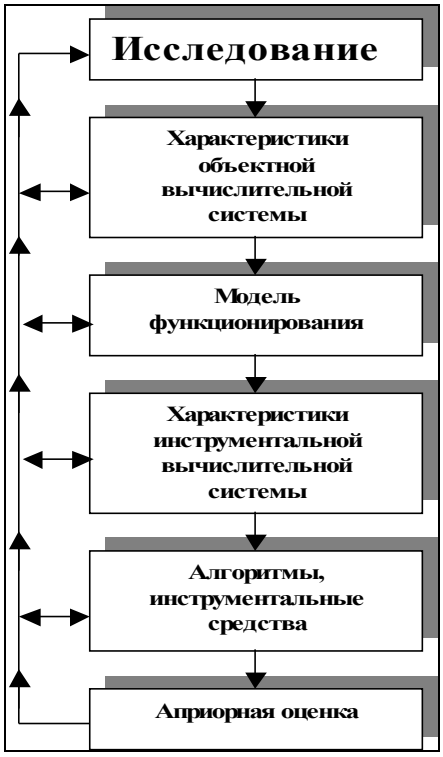
Системы программирования

Система программирование – это комплекс программ, обеспечивающий поддержание жизненного цикла программы в вычислительной системе

Уровень ***системы программирования*** обеспечивает поддержание этапов жизни программы: проектирование, кодирование, тестирование, отладка, изготовление программного продукта.

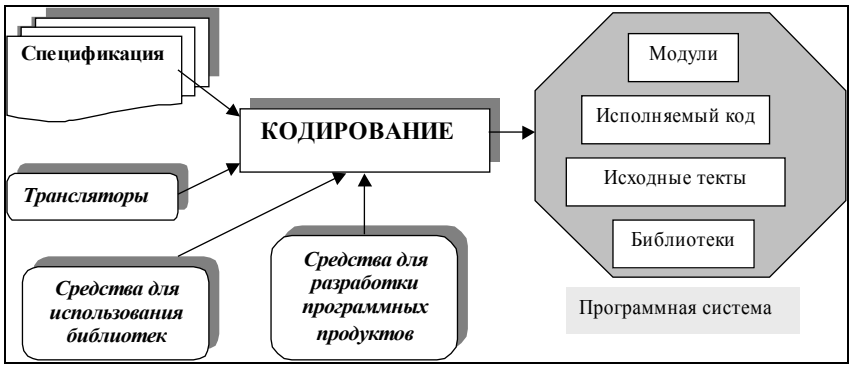
На данном уровне пользователю предоставляются средства программирования виртуальной машины, основанные на некотором языке программирования и совокупности доступных логических/виртуальных ресурсов.

1. Проектирование.



2. Кодирование.

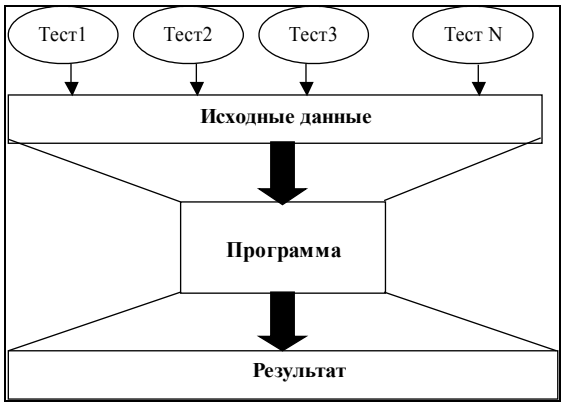
Кодирование – организация поддержки выполнения программы.



Средства для разработки программного обеспечения:

a. средства автоматического контроля межмодульных связей;
b. средства автоматически выполняемых задач;
c. системы поддержки версий.

3. Тестирование и отладка.



Тестирование – проверка спецификации выполнения программы на некоторых наборах данных.

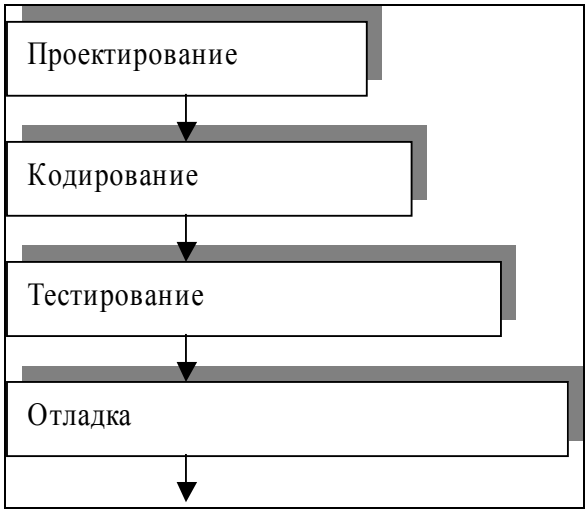
Отладка – процесс локализации ошибок.

4.Ввод программы в эксплуатацию и сопровождение.

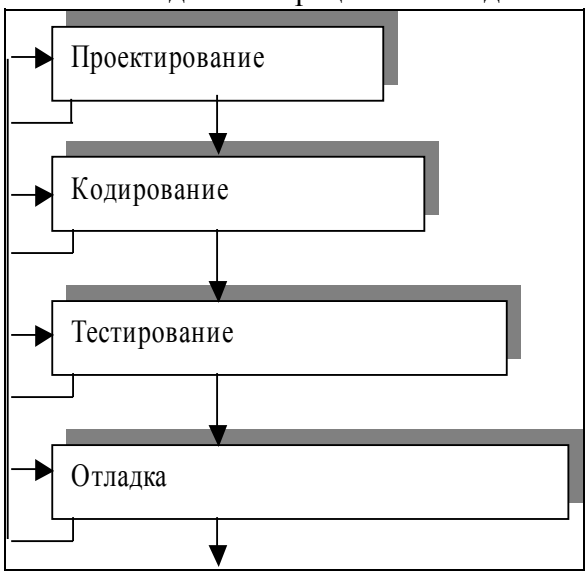
Лекция 2. Системы программирования.

На сегодняшний день существует много различных моделей систем программирования. Основными примерами могут служить каскадная модель (в которой этапы обработки тестирования и отладки работают одновременно, в линейное время, но такая модель почти невозможна, т.к. необходима возможность возврата к предыдущей ступени) и каскадно-итерационная модель (она более реальная, т.к. существует возможность возврата, но более хаотическая, т.к. заказчик не имеет детерминированной информации о сроках получения реальных результатов).

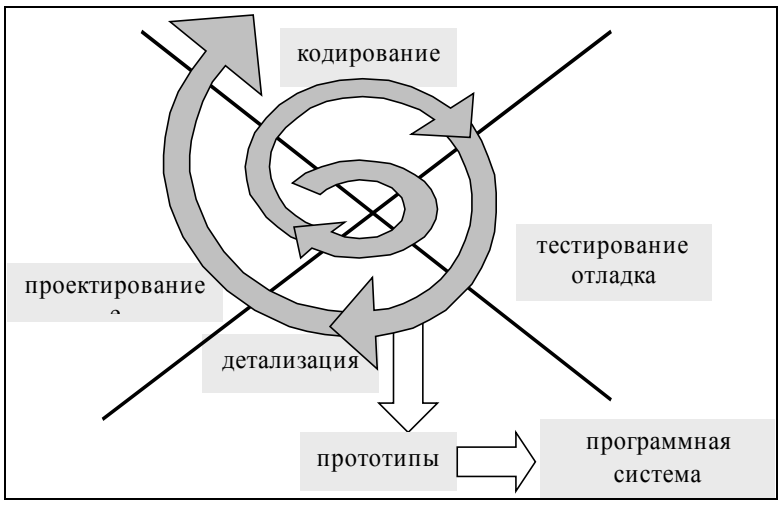
Каскадная модель



Каскадно – итерационная модель



Существует также спектральная модель. Для получения каждого прототипа проходит все этапы, пока какой-нибудь прототип не станет конечным результатом.



Итак, Система программирования – это комплекс программ, обеспечивающий технологию автоматизации

- проектирования,
- кодирования,
- тестирования,
- отладки и сопровождения программного обеспечения.

Этапы развития систем программирования

Начало 50-х годов XX – века.

Система программирования или система автоматизации программирования включала в себя ассемблер (или автокод) и загрузчик, позже появились библиотеки стандартных программ и макрогенераторов.

Середина 50-х – начало 60-х годов XX – века.

Появление и распространение языков программирования высокого уровня (Фортран, Алгол-60, Кобол и др.). Переход от команд-цифр к высокоуровневым программам. Формирование концепций модульного программирования.

Середина 60-х годов – начало 90-х XX – века.

Развитие интерактивных и персональных систем, появление и развитие языков объектно-ориентированного программирования. Появление средств, которые позволяли создавать программные объекты, скрывающие организацию данных и их обработку от пользователя. Появление первых промышленных систем программирования, основанных на языках высокого уровня. Появился язык Си, правда, это скорее машинно-независимый ассемблер.

90-е XX – века – настоящее время.

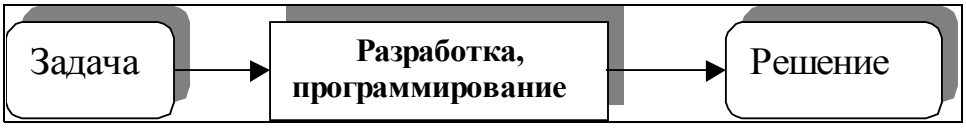
Появление систем программирования, в которых есть комплексные системы. Последовательное взаимозаменение программ. Появление промышленных средств автоматизации проектирования программного обеспечения, CASE-средств (***Computer-Aided Software/System Engineering***), унифицированного языка моделирования UML.

Средства программирования, доступные на уровне системы программирования - программные средства и компоненты СП, обеспечивающие поддержание жизненного цикла программы

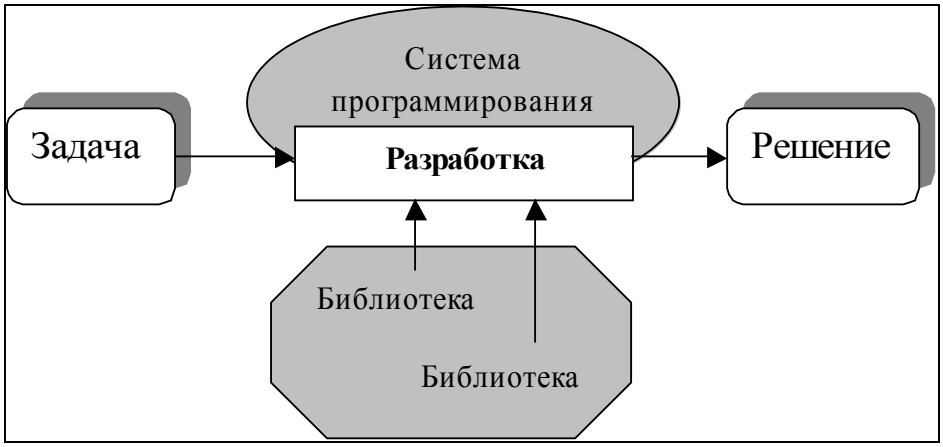
Прикладная система

Прикладная система – программная система, ориентированная на решение или автоматизацию решения задач из конкретной предметной области.

Первый этап развития прикладных систем.

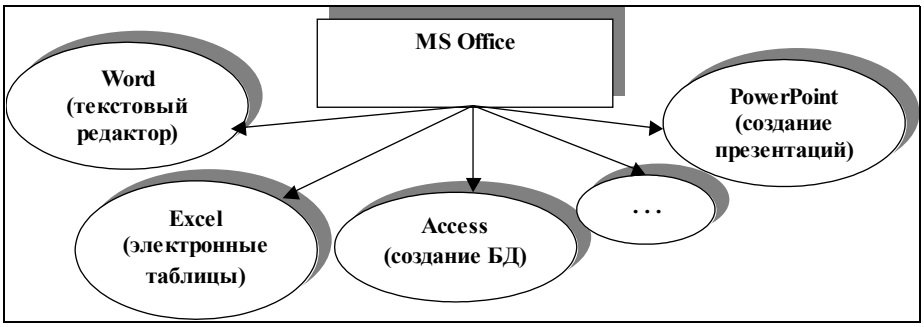


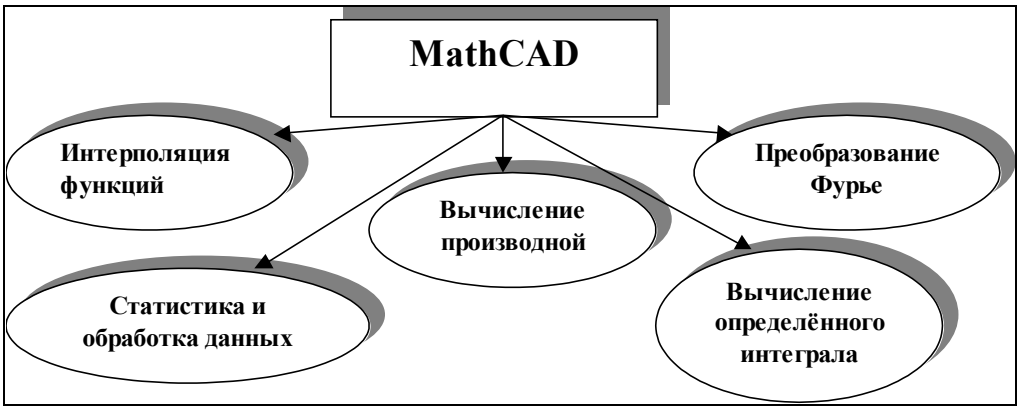
Второй этап – развитие систем программирования и появление средств создания и использования библиотек программ



Третий этап характеризуется появлением *пакетов прикладных программ*

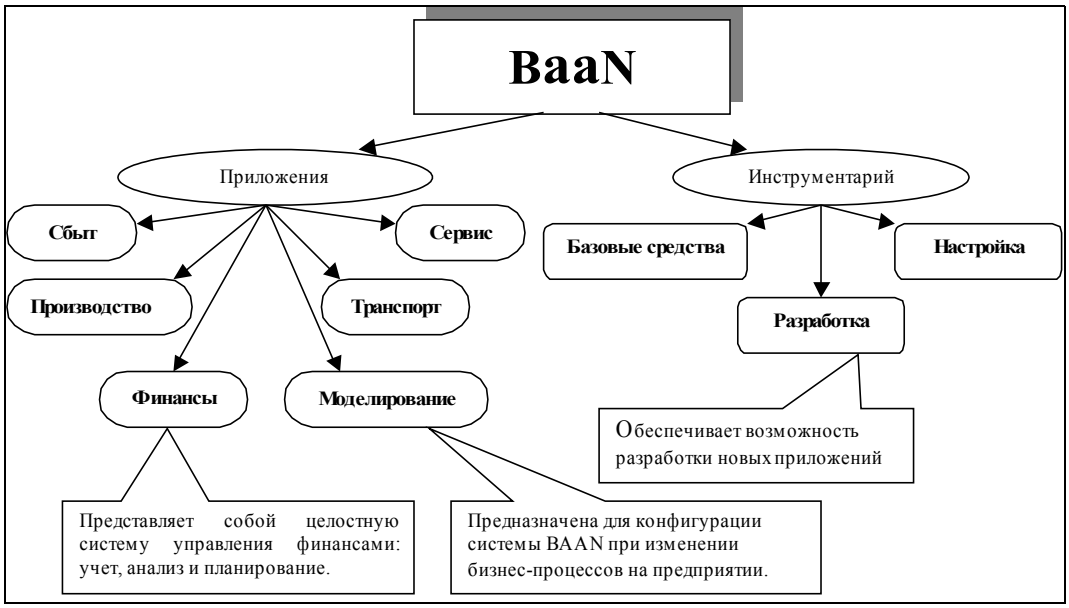
Примеры





Основные тенденции в развитии современных прикладных систем

- ***Стандартизация моделей автоматизируемых бизнес процессов***
- ***B2B*** (business to business)
- ***B2C*** (business to customer)
- ***ERP*** (Enterprise Resource Planning)
- ***CRM*** (Customer Relationship Management)
- ***Открытость системы***
- ***API - Application Programming Interface***



На сегодняшний день специфика прикладных систем – стандартизация бизнесс-процессов. Включается набор средств прикладной системы.

Выводы:

К данному моменту мы обсудили следующие понятия:
- Вычислительная система
- Физические ресурсы (устройства)
- Драйвер физического устройства
- Логические или виртуальные ресурсы (устройства)
- Драйвер логического/виртуального ресурса
- Ресурсы вычислительной системы
- Операционная система
- Жизненный цикл программы в вычислительной системе
- Система программирования
- Прикладная система

К сказанному следует добавить, что пользователь работает в некоторой виртуальной системе. Даже человек, работтающий с Ассемблером, не знает коды операций и команд. К тому же, водной и той же операционной системе может быть целая иерархия виртуальных систем.

Основы компьютерной архитектуры.

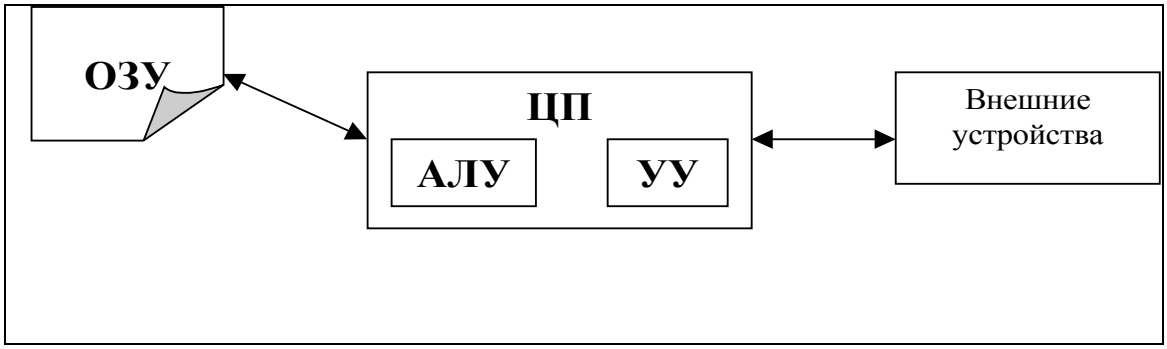
Невозможно говорить отдельно об ОС или об архитектуре компьютеров, т.к. они взаимодействуют и сильно интегрированы. Их эффективность зависит от ОП и HardWare компьютера.

Джон фон Нейман(John Von Neumann). С этим именем связана машина фон Неймана. Он опубликовал Предварительный доклад о компьютере EDVAC (A First Draft Report on the EDVAC), в котором отразил основные концепции организации компьютера. EDVAC (Electronic Discrete Variable Computer - Электронный Компьютер Дискретных Переменных)
. Основные разработчики этой модели - Джон Мочли (John Mauchly) и Джон Преспер Эккерт (John Presper Eckert), авторы
ENIACa (Electronic Numerical Integrator And Computer).

Принципы Фон Неймана:

- принцип двоичного кодирования. Все данные кодируются с помощью двоичных сигналов.
- принцип программного управления. Программы состоят из команд, в которых закодированы операции и операнды. Выполнение компонент программы – автоматическое выполнение команд, составляющих программу. Последовательность команд определяется последовательностью команд и данных.
- Принцип хранения программ. Для команд и данных единое устройство памяти. Все слова имеют последовательную адресацию. Интерпретации информации, размещенной в памяти, происходит в момент работы с данной ячейкой памяти.

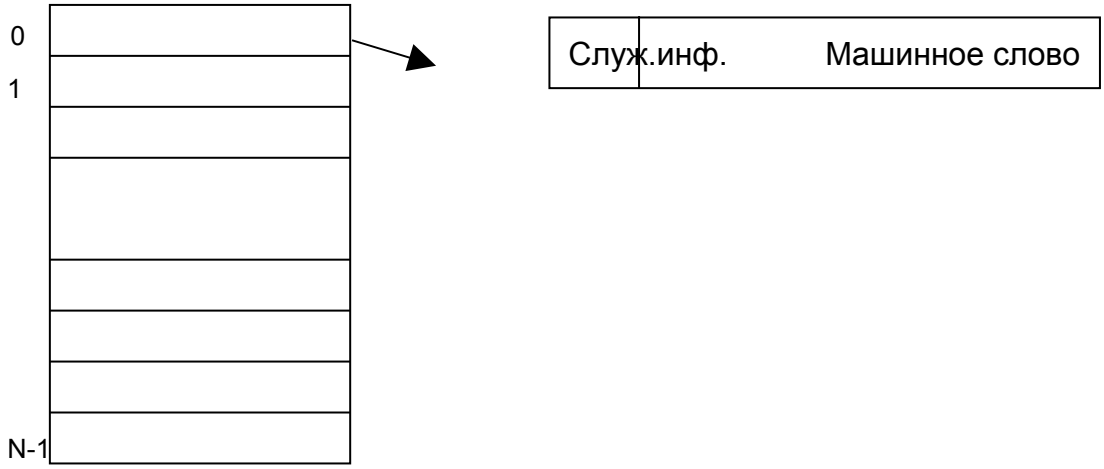
Предполагается, что компьютер фон Неймана имеет следующие компоненты:



Оперативное запоминающее устройство (основная память, оперативная память(ОП))

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.

RAM – устройство произвольного доступа.



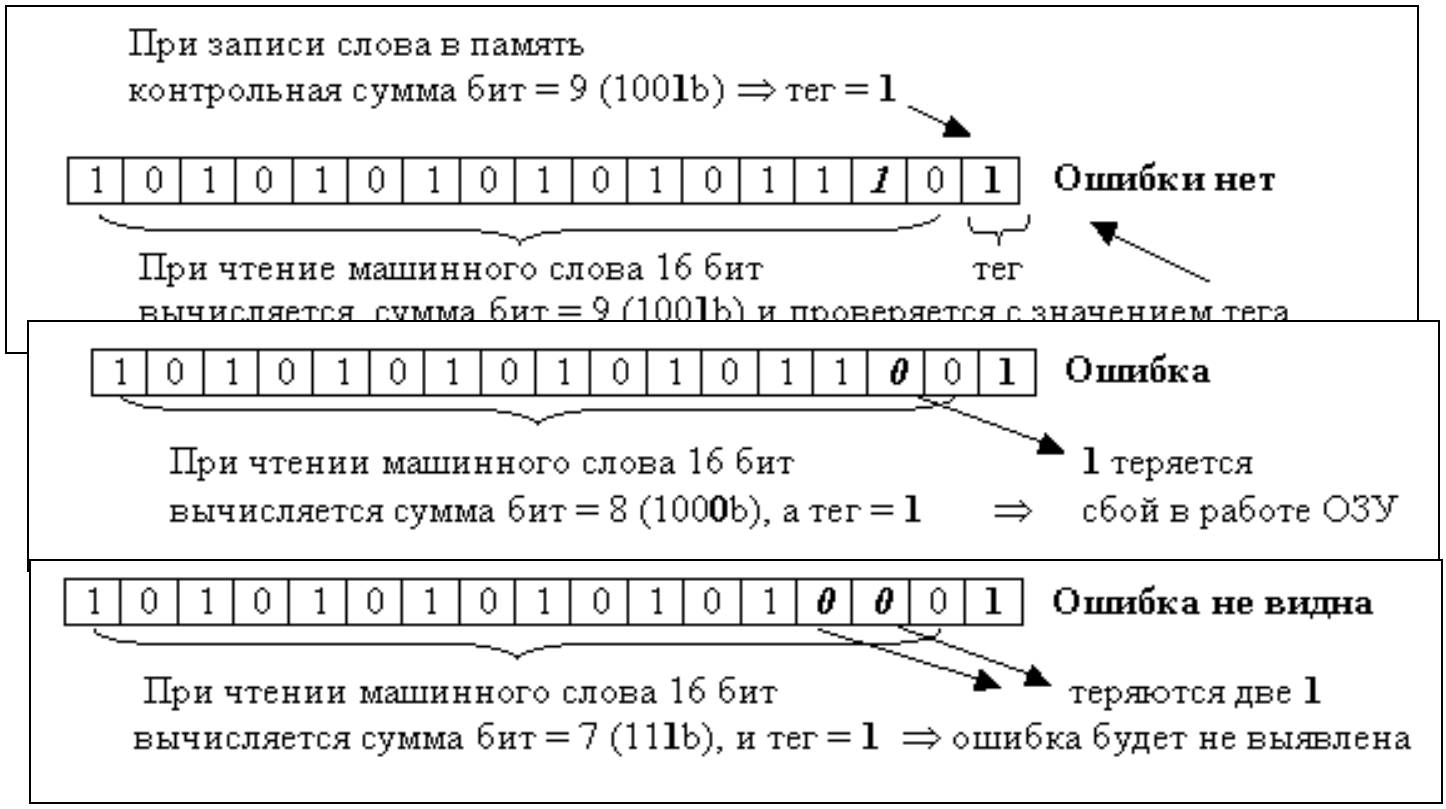
Машинное слово – поле программно изменяемой информации, в машинном слове могут располагаться машинные команды (или части машинных команд) или данные, с которыми может оперировать программа. Машинное слово имеет фиксированный для данной ЭВМ размер (обычно размер машинного слова – это количество двоичных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

Пример контроля за целостностью данных по четности



- 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.

Важная характеристика ОП - производительность - скорость доступа процессора к данным, размещенным в ОЗУ:

•**время доступа (access time- t_{access}) -** время между запросом на чтение слова из оперативной памяти и получением содержимого этого слова.

•**длительность цикла памяти (cycle time - t_{cycle}) -** минимальное время между началом текущего и последующего обращения к памяти.

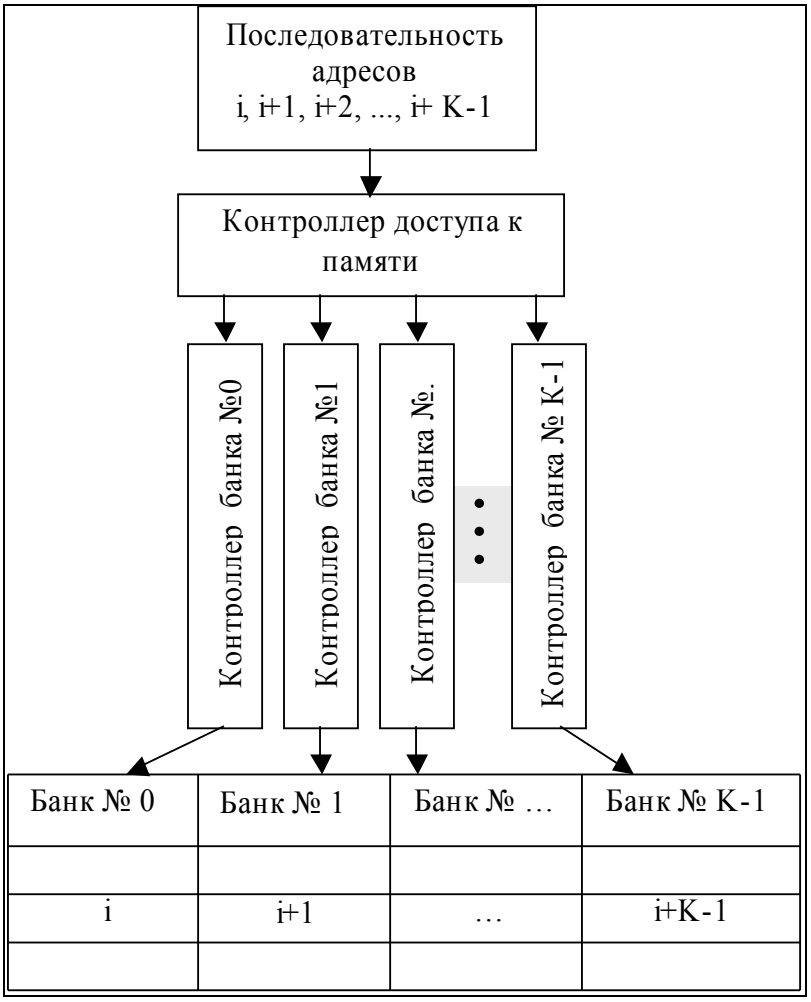
•

$$(t_{cycle} > t_{access})$$

Расслоение памяти

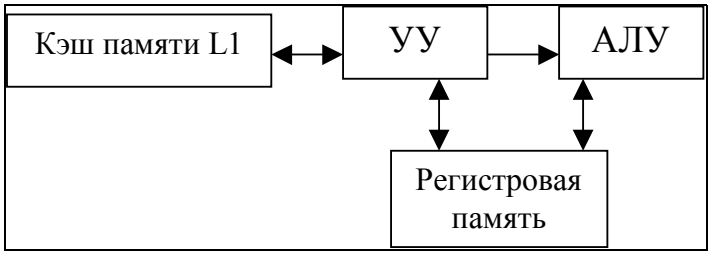
ОЗУ делится на **К** независимых банков памяти, где $\mathbf{K} = 2^L$

Последовательное чтение машинных	ОЗУ без расслоения	ОЗУ с расслоением
A	$\sim t_{cycle}$	$\sim t_{access}$
A+1	$\sim t_{cycle}$	$\sim t_{access}$
A+2	$\sim t_{cycle}$	$\sim t_{access}$
Суммарное время:	$\sim 3*t_{cycle}$	$\sim 3*t_{access}$


Последовательность адресов
i, i+1, i+2, ..., i+ K-1
Контроллер доступа к памяти
Контроллер банка №0
Контроллер банка №1
Контроллер банка №.
Контроллер банка № K-1
Банк № 0
Банк № 1
Банк № …
Банк № K-1
i
i+1
…
i+K-1

Центральный процессор

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:

- 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.

- 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.

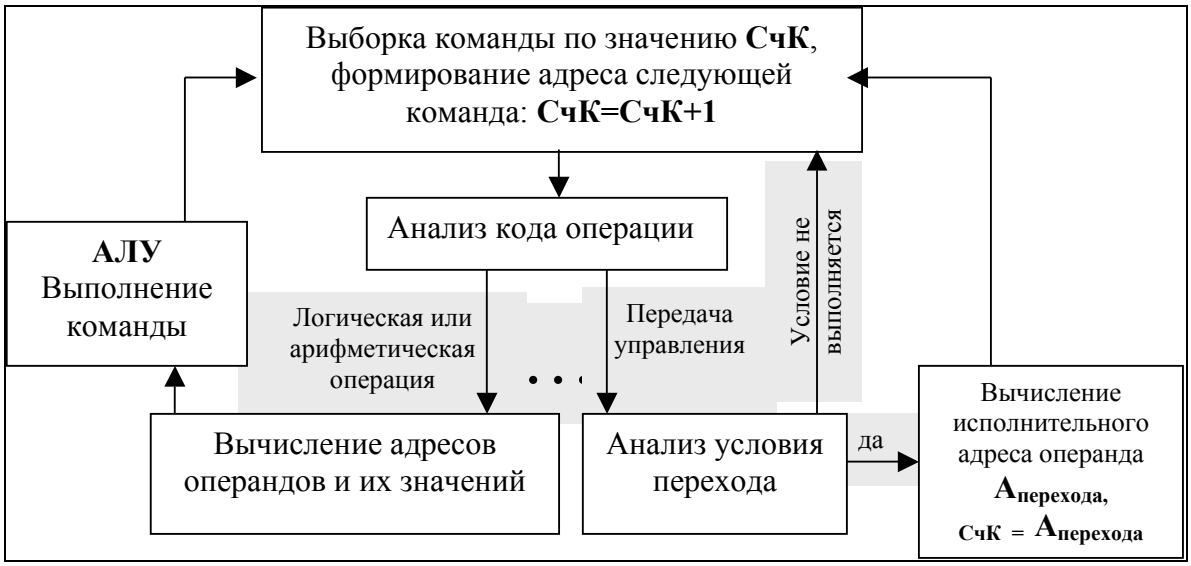
К специальным относятся:

- регистр адреса
- регистр результат
- регистры внешних устройств
- регистры – указатели стека

Устройство управления (control unit)– координирует выполнение команд программы процессором.

Арифметико-логическое устройство (arithmetic/logic unit) – обеспечивает выполнение команд, предусматривающих арифметическую или логическую обработку операндов.

Рабочий цикл процессора.



КЭШ память

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.

Суть КЭШа - Обмен данными между КЭШем и оперативной памятью осуществляется блоками фиксированного размера.

Каждому буферу КЭШа соответствует адресный тег блока, который содержит служебную информацию о блоке (соответствие области ОЗУ, свободен/занят блок, ...).

Когда УУ вычисляет исполнительный адрес операнда, контролер памяти обращается к аппаратной системе КЭШа. Нахождения данных в КЭШе - попаданием (hit). Если искомых данных нет в КЭШе, то фиксируется промах (cach miss).

При возникновении промаха происходит обновление содержимого КЭШа - вытеснение. Стратегии вытеснения:

- •случайная;
- •вытеснение наименее популярного (LRU - Least-Recently Used

Сквозное кэширование (write-through caching)- расслоение памяти на блоки, работающие параллельно,

кэширование с обратной связью (write-back cache) - тег модификации (dirty bit)

Аппарат прерываний

Прерывание - событие в компьютере, при возникновении которого в процессоре происходит предопределенная последовательность действий.

При выполнении программы может произойти возникновение некоторых исключительных или критических ситуаций (сломалось устройство, произошло деление на ноль). Для этого в аппаратуре фиксируется набор событий, на возникновение каждого из которых машина реагирует предопределенным образом. Эти события называются прерываниями, а реакция на эти события - обработка прерываний.

Прерывания делятся на два типа:

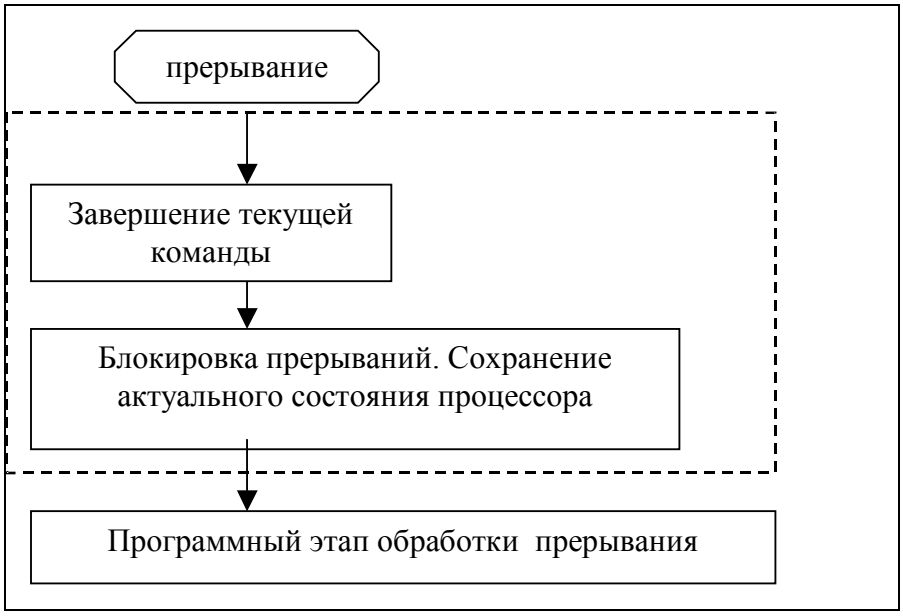
1) внутренние - возникают в схемах контроля процессора. Примеры внутренних прерываний: произошло переполнение (overflow) или деление на ноль.
2) внешние - наступают во «внешнем мире», во внешних устройствах, поступают из УУВУ. Эти прерывания связаны с событием (часто это бывает просто ошибка), произошедшем вовне (например, невозможно считать данные с HDD).

Рассмотрим теперь последовательность действий по обработке прерываний:

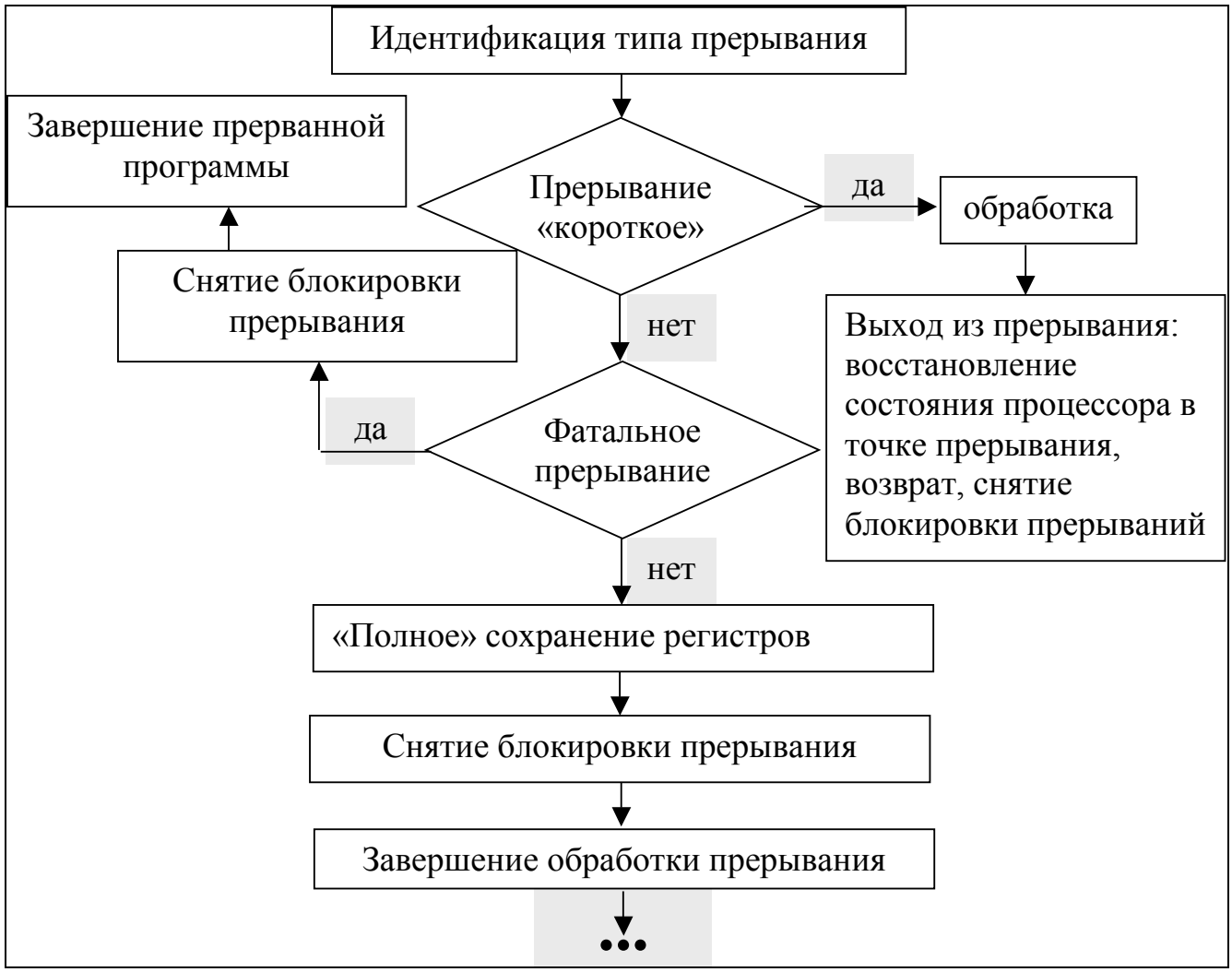
Сразу заметим, прерывание не обязательно должно вызвать прекращение программы. После обработки такого прерывания система должна иметь возможность продолжить работу программы. Например, по завершении обмена с HDD происходит прерывание, но это не значит что после этого работа программы должна завершиться.

1) Итак, первое, что делает аппаратура при возникновении прерывания - это так называемое «малое упрятывание» текущей информации о программе. Аппаратура «прячет» в специальные регистры минимальный набор информации о выполняемой программе. Обычно, в этот набор данных входят значение регистра-счетчика команд (IP), содержимое регистра результата, указатель стека и несколько регистров общего назначения, которыми будет пользоваться операционная система (ОС).
2) В некоторый специальный управляющий регистр, условно будем его называть регистром прерываний, помещается код возникшего прерывания.
3) Запускается программа обработки прерываний операционной системы, т.е. передается управление на некоторую фиксированную точку ОС. (Замечу, что здесь в зависимости от реализации имеется две возможности: либо точка одна - тогда тип прерывания передается через параметр, либо для каждого прерывания имеется своя точка).
4) Происходит анализ причин прерывания. При этом используются только «упрятанные» (сохраненные) регистры. Если это прерывание было фатальным (деление на ноль, например), то продолжать выполнение программы не имеет смысла и управление передается на ту часть ОС, которая завершит выполнение программы. Если же это прерывание не фатальное, то происходит дополнительный анализ, который приводит к ответу на вопрос: можно ли оперативно (быстро) обработать прерывание. Пример прерывания, которое можно всегда обработать оперативно - прерывание по таймеру. Прерывание, связанное с приходом информации по линии связи, нельзя обработать оперативно – в этом случае происходит расчищение в системе места для программы ОС, которая займется обработкой этого прерывания. Т.е. при невозможности оперативной обработки прерывания происходит так называемое «полное упрятывание» - сохранение спасенных регистров, а затем и всех остальных регистров в таблице ОС (а не в аппаратных регистрах!). Затем фиксируется тот факт, что пространство ОЗУ, занимаемое программой, может быть перенесено (при необходимости) на внешнее устройство. Дальше идет обработка прерывания, затем происходит восстановление значений регистров и осуществляется возврат в программу в ту же точку, на которой программа остановилась.

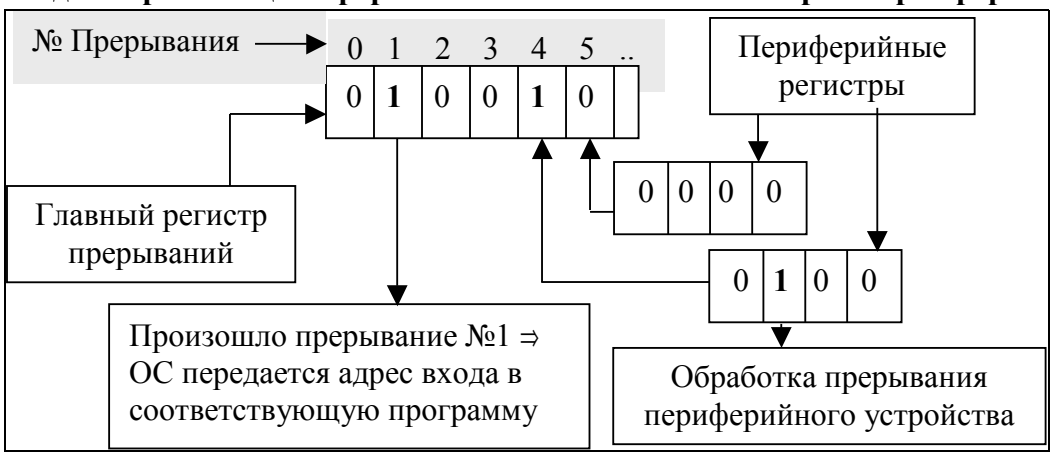
Этап аппаратной обработки прерываний



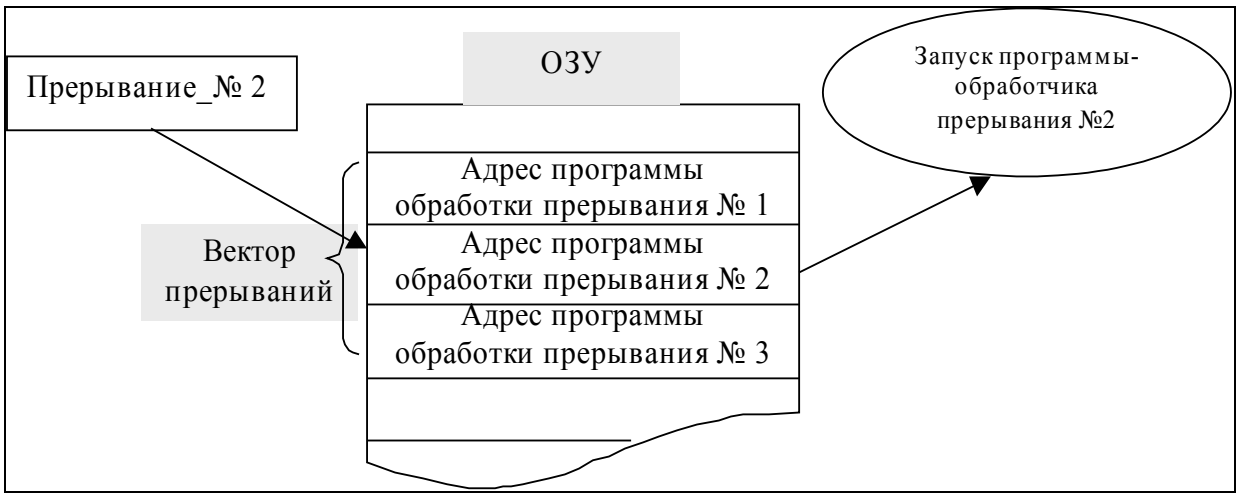
Программный этап обработки прерываний



Модель организации прерываний с использованием «регистра прерываний»



Модель организации прерываний с использованием «вектора прерываний»


Прерывание_№ 2
ОЗУ
Запуск программы-обработчика прерывания №2
Адрес программы обработки прерывания № 1
Адрес программы обработки прерывания № 2
Адрес программы обработки прерывания № 3
Вектор прерываний

Лекция 3.Внешние запоминающие устройства.

Система построена из разнообразных элементов, поэтому для того, чтобы она правильно функционировала, применяются специальные средства, сглаживающие дисбаланс (историческое решение – регистры общего назначения), достигается минимум реальной частоты обращения к памяти - расслоение памяти (это параллельность передачи данных и передача управления). Считывание в память фрагментов данных – организация ассоциативной памяти – КЭШа – сверхоперативной памяти, которая аккумулирует наиболее частые команды и минимизирует обращения к оперативной памяти. Также широко используется аппарат прерываний- средство, аппаратно-ориентированное на своевременную обработку поступающей информации, осуществляющее реакцию на ошибки.

Система внешних запоминающих устройств (ВЗУ) компьютера очень широка, она включает в себя типовой набор внешних устройств Традиционно это:

- внешние запоминающие устройства, предназначенные для организации хранения данных и программ;
- устр-ва ввода и отображения, предназначенные для ввода извне данных и программ и отображения этой инф-ции;
- устр-ва приема данных, предназначенные для получения данных от других компьютеров и извне.

1) Обмен данными:
• записями фиксированного размера – *блоками*
• записями произвольного размера
2) Доступ к данным:
• операции чтения и записи (жесткий диск, CDRW).
• только операции чтения (CDROM, DVDROM, …).

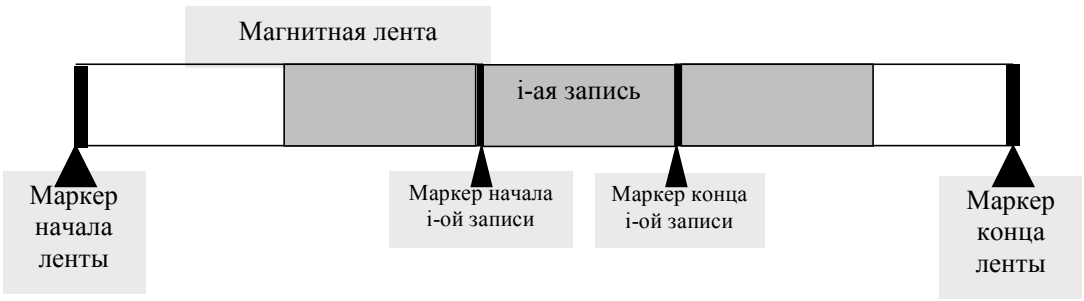
По доступу к данным различают устройства:

- **Последовательного доступа:**

  В устройствах последовательного доступа для чтения i-ого блока памяти необходимо пройти по первым i-1 блокам. (Пример: пульт дистанционного управления ТВ, кнопки «канал+» и «канал-» или бытовые кассетные магнитофоны - ВЗУ большинства бытовых компьютеров 80-х годов). Устройства последовательного доступа менее эффективны чем устройства прямого доступа.

  •Магнитная лента

  Это тип ВЗУ широко известный всем пользователям бытовых компьютеров. Его



принцип основывается на записи/чтении информации на магнитной ленте. При этом перед

началом и окончанием очередной записи на ленту заносится специальный признак - маркер начала, соответственно, конца. Головка проходит на нужный номер цилиндра и не более чем зза 1 оборот находит нужный сектор.

- **Прямого доступа:**

В устройствах прямого доступа для того, чтобы прочесть i-ый блок данных, не нужно читать первые i-1 блоков данных. (Пример: пульт дистанционного управления Вашего телевизора, кнопки - каналы; для того, чтобы посмотреть i-ый канал не нужно «перещелкивать» первые i-1).

• Магнитные диски

Конструкция ВЗУ данного типа состоит в том, что имеется несколько дисков (компьютерный жаргон - «блины»), обладающих возможностью с помощью эффекта перемагничивания хранить информацию, размещенных на оси, которые вращаются с некоторой постоянной скоростью. Каждый такой диск имеет одну или две поверхности, покрытые слоем, позволяющим записывать информацию.

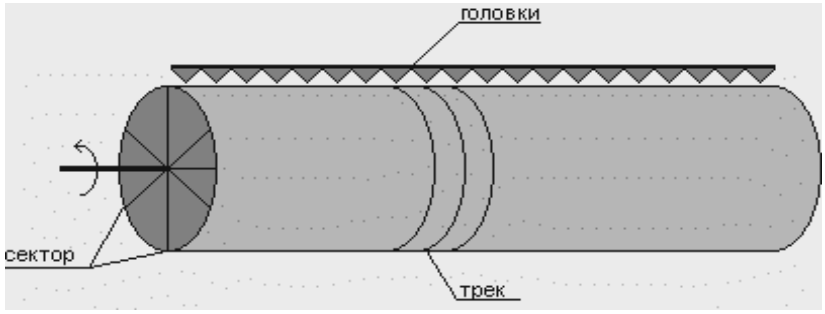
Диски имеют номера; поверхности каждого диска также пронумерованы (0-ая поверхность, 1-ая поверхность). Концентрическим окружностям одного радиуса на каждом диске соответствует условный цилиндр. (А каждая такая окружность - дорожка.)

Диск также разбит на равные сектора. Информация на диске адресуется с помощью 4-х координат: № диска, № поверхности, № цилиндра, № сектора.

Штанга имеет механически перемещаемые щупы, на концах которых находятся считывающие и записывающие головки. В большинстве конструктивных решений количество этих щупов равно количеству дисков (считывается либо верхняя, либо нижняя поверхность).

Обмен информацией осуществляется по следующей схеме: на блок управления диском поступает набор координат и размер требуемого блока информации. Затем, включается головка, читающая заданную поверхность заданного диска. Она подводится к нужному цилиндру и ожидает подхода нужного сектора. После этого осуществляется обмен.

• Магнитный барабан



Имеется металлический цилиндр большой массы, вращающийся вокруг своей оси. Роль большой массы - поддержание стабильной скорости вращения. Поверхность этого цилиндра покрыта магнитным слоем, позволяющим хранить, читать и записывать информацию. Поверхность барабана разделена на n равных частей (в виде колец), которые называются треками (track). Над барабаном расположен блок неподвижных головок так, что над каждым треком

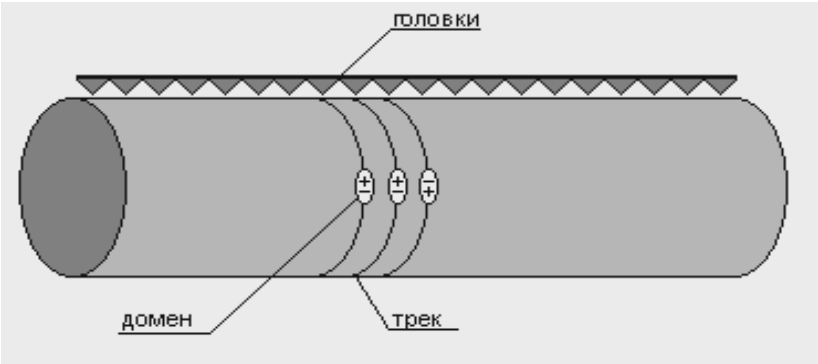
расположена одна и только одна головка. (Головок, соответственно, тоже n штук). Головки способны считывать и записывать информацию с барабана. Каждый трек разделен на равные сектора. В каждый момент времени в устройстве может работать только одна головка. Запись информации происходит по трекам барабана, начиная с определенного сектора. При заказе на обмен поступают следующие параметры: № трека, № сектора, объем информации.

При чтении информации происходят следующие действия. Включается головка,

соответствующая заданному номеру трека, происходит ожидание того момента, когда над головкой окажется нужный сектор. После этого происходит обмен. Магнитные барабаны работают быстрее чем магнитные диски ввиду того, что в них используется меньше «механики». Но это довольно габаритный аппарат, его запуск требует довольно много времени для того, чтобы скорость вращения дисков стабилизировалась и стала постоянной.

• Магнито-электронные ВЗУ прямого доступа (память на магнитных доменах)

В конструкции этого устройства опять таки используется барабан, но на этот раз он неподвижен. Барабан опять таки разделен на треки и над каждым треком имеется своя головка. За счет некоторых магнитно-электрических эффектов происходит перемещение по треку цепочки доменов. При этом каждый домен однозначно ориентирован, то есть он бежит стороной, с зарядом «+», либо стороной, заряженной «-». Так кодируется ноль и единица. Эта память очень быстродейственна, т.к. в ней нет «механики». Память на магнитных доменах очень дорога и используется в большинстве случаев в военной и космической областях.



Обмен между внешними устройствами происходит с помощью

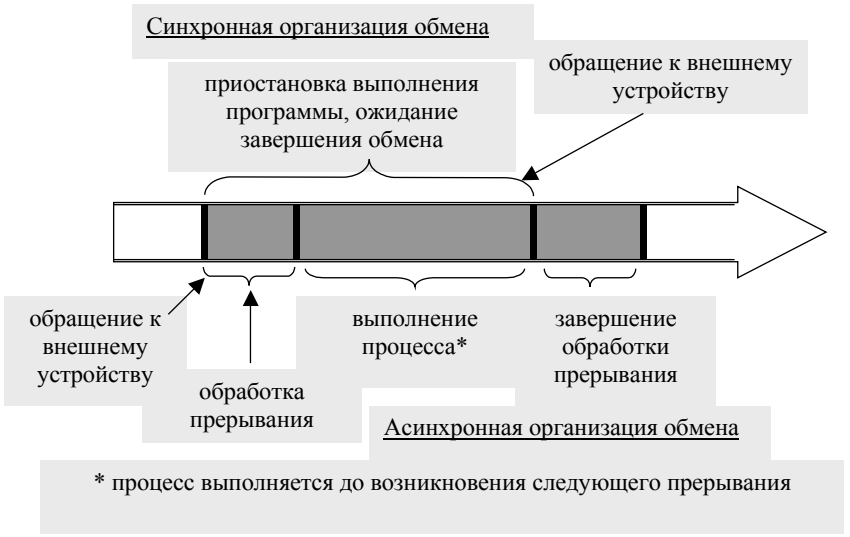
- потока управляющей информации
- потока данных

Организация уравления такова:

1. **Непосредственное управление** внешними устройствами центральным процессором.

2. **Синхронное** управление внешними устройствами с использованием контроллеров внешних устройств
3. **Асинхронное** управление внешними устройствами с использованием контроллеров внешних устройств

ЦП подает управляющую инф-цию по организации обмена, но поток образующихся данных идет между устройствами и ОП, без ЦП.



4. Использование **контроллера прямого доступа** к памяти (DMA) при обмене.

5. Управление внешними устройствами **с использованием процессора или канала ввода/вывода**.

Обеспечивается высокоуровневый интерфейс для организации ввода\вывода.

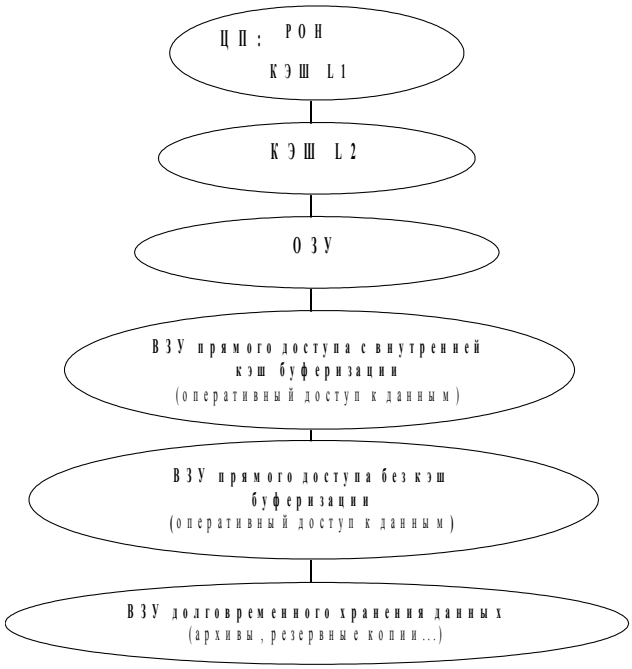
6. Использование **контроллера прямого доступа** к памяти (DMA) при обмене.

ЦП подает управляющую инф-цию по организации обмена, но поток образующихся данных идет между устройствами и ОП, без ЦП.

7. Управление внешними устройствами **с использованием процессора или канала ввода/вывода**.

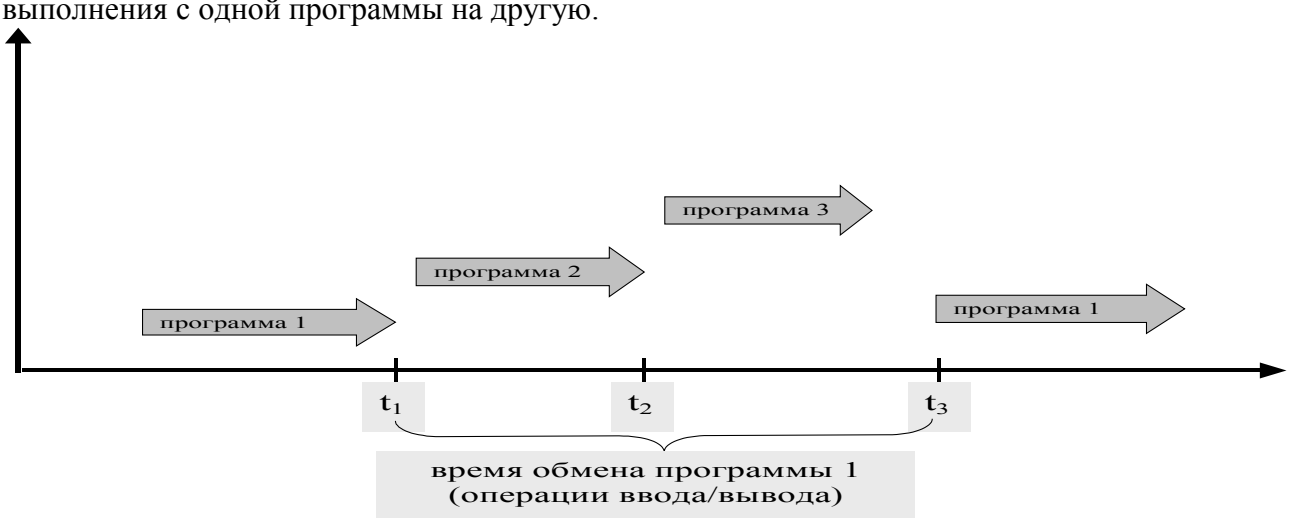
Обеспечивается высокоуровневый интерфейс для организации ввода\вывода.

Иерархия памяти
Выглядит следующим образом:



Аппаратная поддержка ОС и систем программирования.

Мультипрограммный режим : режим при котором возможна организация переключения выполнения с одной программы на другую.



Аппаратные средства компьютера, необходимые для поддержания мультипрограммного режима:

1. Аппарат **защиты памяти**.
2. **Специальный режим операционной системы.** (привилегированный режим или режим супервизора)
3. Аппарат прерываний (как минимум, **прерывание по таймеру**).

Некоторые проблемы

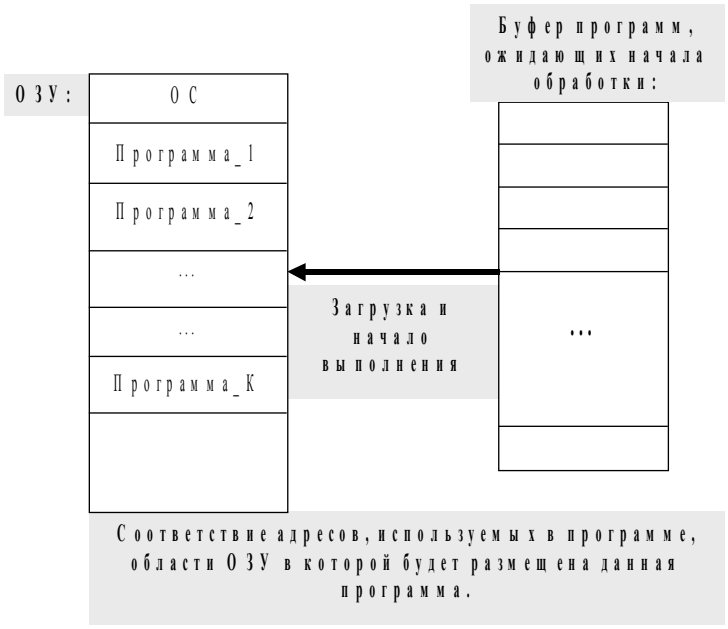
1. **Вложенные обращения к подпрограммам**

2. Накладные расходы при смене обрабатываемой программы:

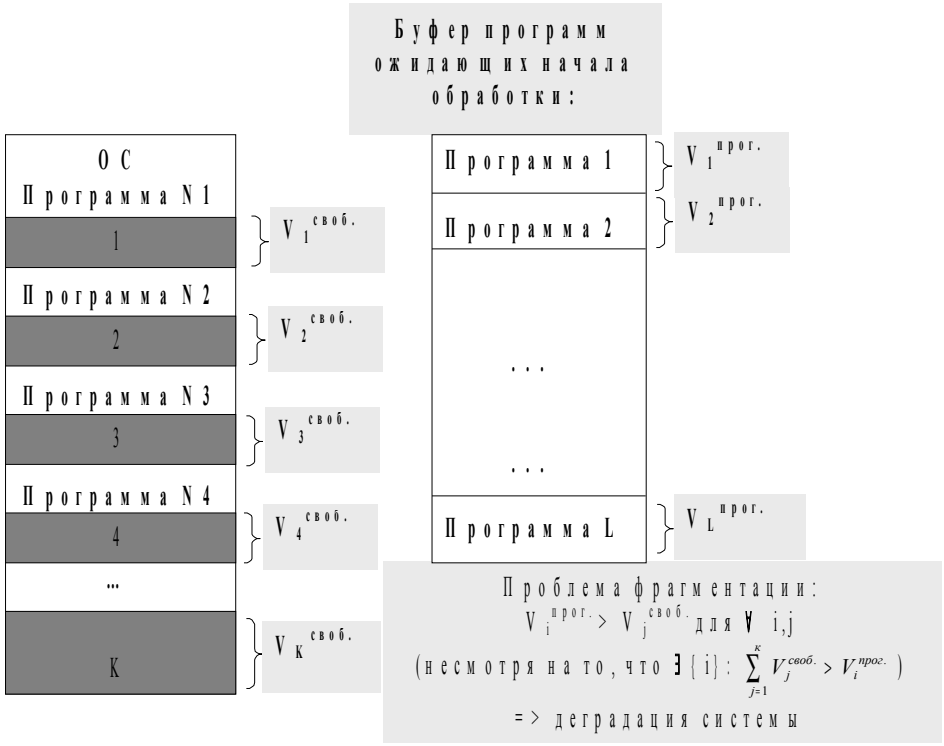
• необходимость включения режима блокировки прерываний;

• программное сохранение / восстановление содержимого регистров при обработке прерываний;

3. **Перемещаемость программы по ОЗУ**

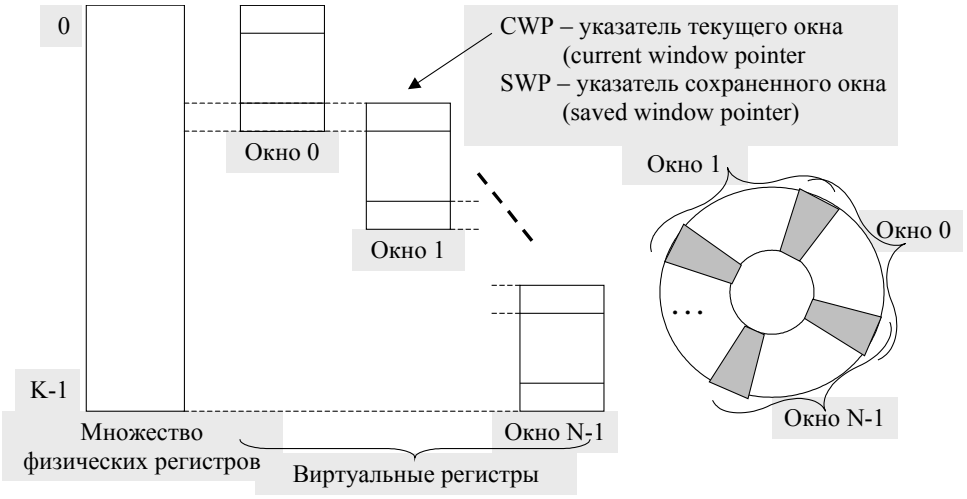


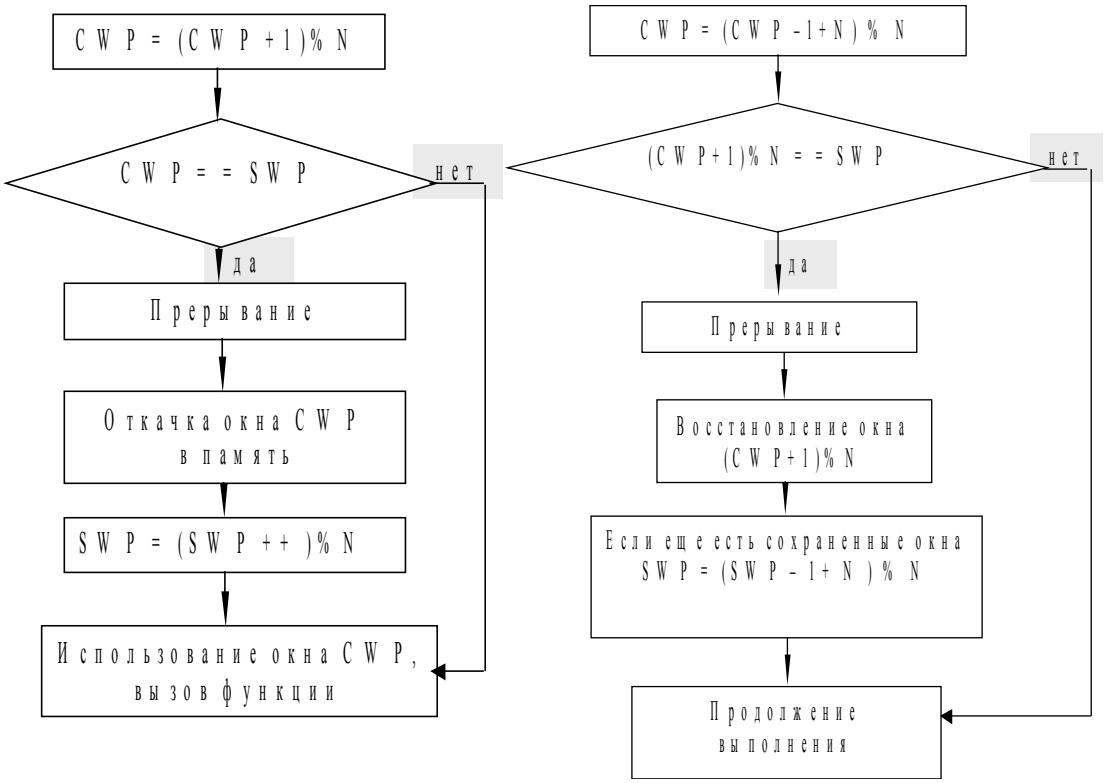
4. Фрагментация памяти



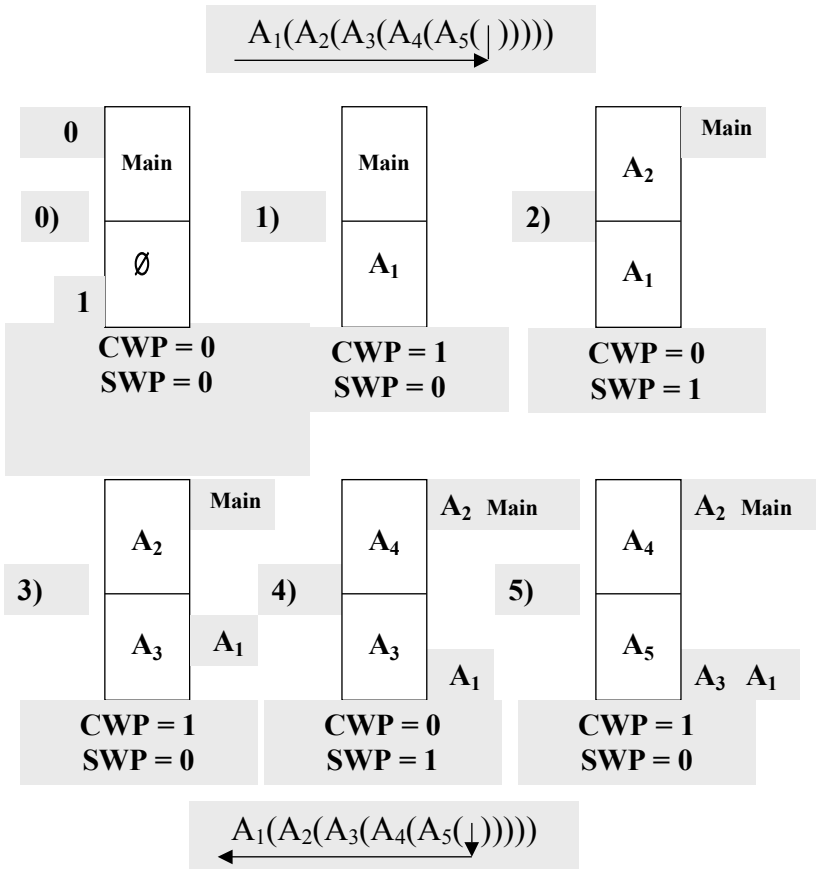
4.

Регистровые окна

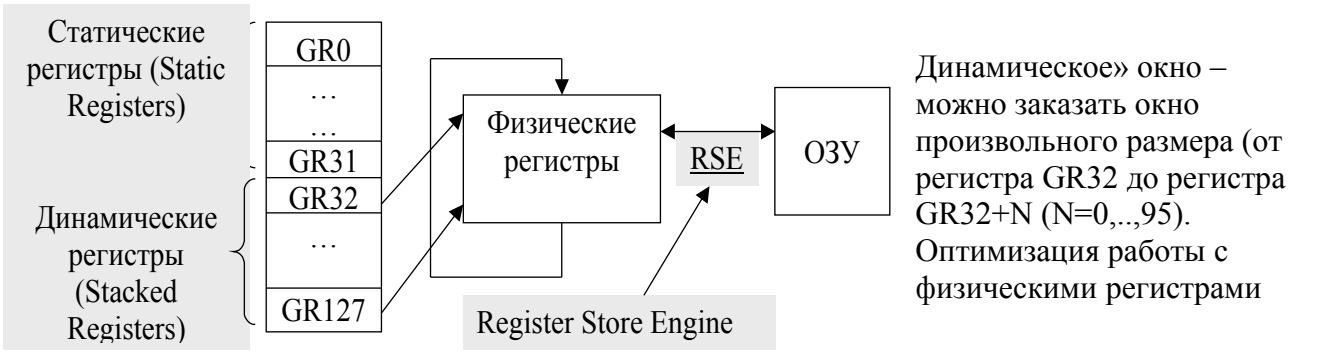




Регистровые окна (register window)
Пример:

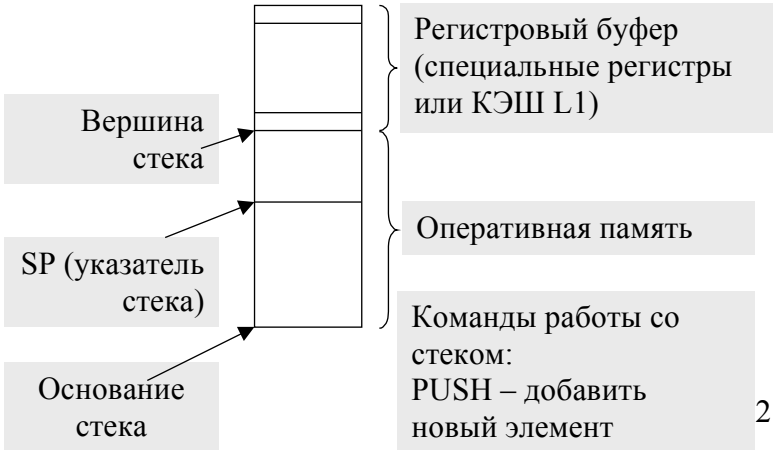


Модель организации регистровой памяти в Intel Itanium



системный стек

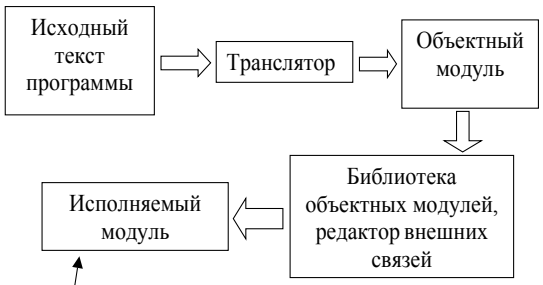
Использование системного стека может частично решать



проблему минимизации накладных расходов при смене обрабатываемой программы и/или обработке прерываний.

Виртуальная память

1. базирование адресов



Программная (логическая или виртуальная) адресация

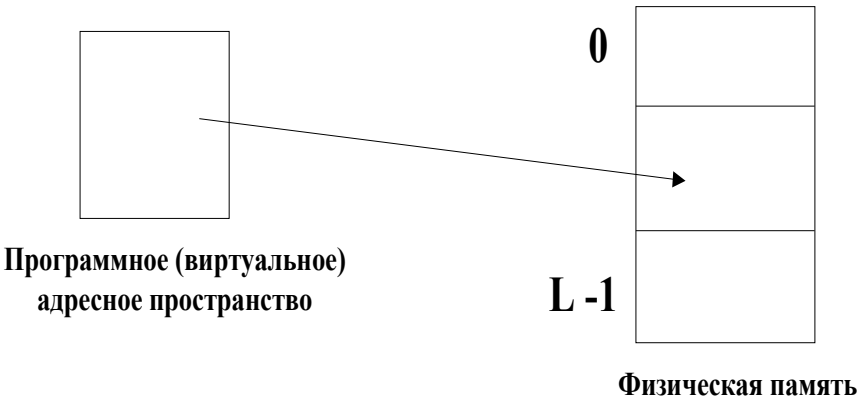
Проблема – установление соответствия между программной адресацией и физической памятью

Аппарат виртуальной памяти – аппаратные средства компьютера, обеспечивающие преобразование (установление соответствия) программных адресов, используемых в программе адресам физической памяти в которой размещена программа при выполнении.

Базирование адресов – реализация одной из моделей аппарата виртуальной памяти.

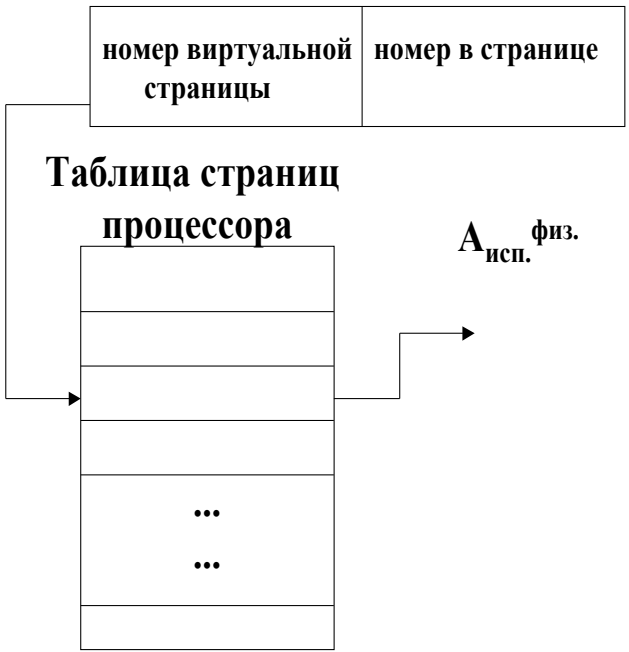
Базирование адресов – средство отображения виртуального адресного пространства программы в физическую память «один в один».

Базирование памяти решает проблему перемещения, но не решает проблему фрагментации. Для решения проблемы фрагментации используются более развитые механизмы организации ОЗУ и виртуальной памяти



2. страничная память

Память аппаратно разделена на блоки фиксированного размера – страницы. Размер страницы – 2^k



Структура адреса:

к к-1

0

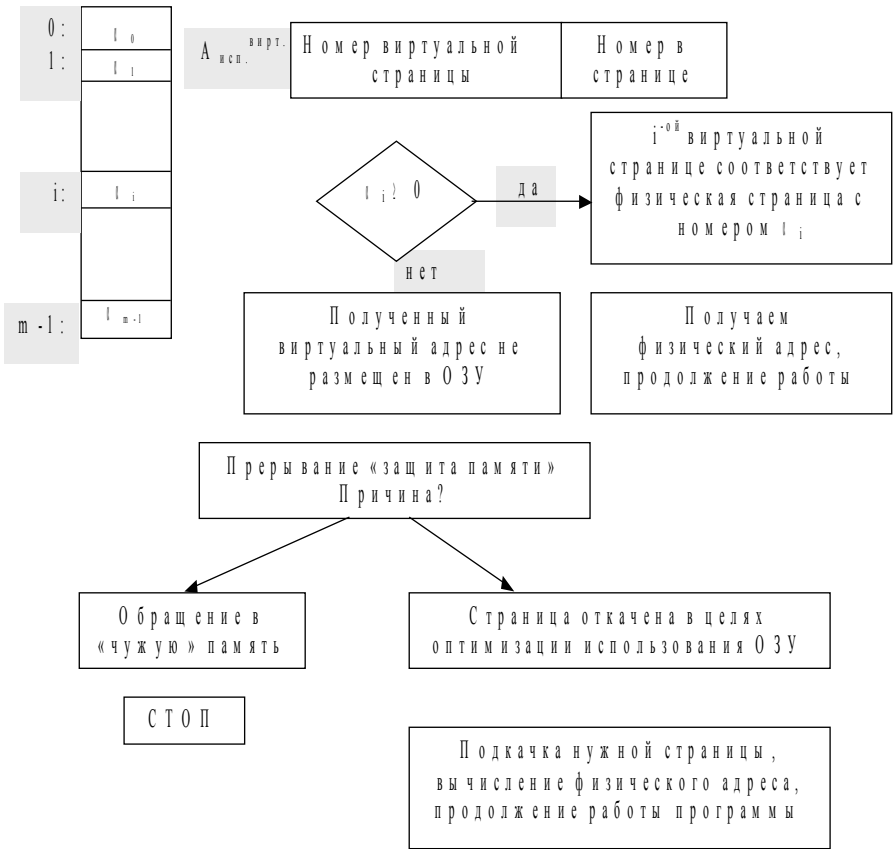
Количество страниц ограничено размером поля «номер страницы

Преобразование виртуального адреса в физический – замена номера виртуальной страницы на соответствующий номер физической страницы

Виртуальное адресное пространство – множество виртуальных страниц, доступных для использования в программе. Количество виртуальных страниц определяется размером поля «номер виртуальной страницы» в адресе.

Физическое адресное пространство – оперативная память, подключенная к данному компьютеру. Физическая память может иметь произвольный размер (число физических страниц может быть меньше, больше или равно числу виртуальных страниц).

Модельный пример организации страничной виртуальной памяти.



Лекция 4.Операционная система. Общие характеристики и свойства.

Операционная система – это комплекс программ, обеспечивающий контроль за существованием, распределением и использованием ресурсов ВС.
Расшифруем:

- существование – логические и виртуальные ресурсы существуют за счет распределения и использования ресурсов ОС;

- распределение – современные ОС всегда многопроцессорные;
- использование – проблема организации контроля за использованием, например, учет времени ЦП.

Процесс – это одно из базовых понятий. Синонимами процесса являются управление задачами, заданиями. Каждая ОС рассматривает некоторую сущность, которую мы называем процессом. Это элементарная единица, которая осуществляет управление ОС. Это совокупность машинных команд и данных, исполняющаяся в рамках ВС и обладающая правами на владение некоторым набором ресурсов.

Важнейшее св-во процесса – набор ресурсов:

- монопольно принадлежащих данному процессу;
- разделяемых (принадлежащих 2м или более процессам).

В настоящее время проблема организации разделяемых ресурсов довольно существенна.
Существуют 2 модели:

1) предварительная декларация необходимых ресурсов (объем памяти, номера виртуальных страниц => запуск процесса);
2) модель динамического дополнения списка ресурсов (запуск процессов, в которых нужен только стартовый объем пакета, а остальное выделяется по необходимости.

Достоинства: Первая модель более строгая, а вторая более гибкая.

В независимости от того, какие функции способна выполнять ОС, она должна удовлетворять пяти основным свойствам:

1. **Надежность.** ОС должна быть, по меньшей мере, так же надежна, как аппаратура, на которой она работает. В случае возникновении ошибки в программном или аппаратном оборудовании система должна обнаружить эту ошибку и либо попытаться исправить положение, либо, по крайней мере, постараться свести к минимуму ущерб, нанесенный этой ошибкой пользователям.
2. **Защита.** Имеется в виду защита и персонификация данных пользователей. Этой темы мы коснемся позже.
3. **Эффективность.** Обычно ОС представляет собой сложную программу, которая использует значительную часть аппаратных ресурсов для своих собственных надобностей. Ресурсы, которые потребляет ОС, не поступают в распоряжение пользователей. Следовательно, сама система должна быть как можно более экономной. Кроме того, система должна управлять ресурсами пользователей так, чтобы добиться максимальной загруженности ресурса в те доли времени, когда ресурс не простаивает (В большей степени это касается загруженности процессора (процессоров)).

4. **Предсказуемость.** Требования, которые пользователь может предъявить к системе, в большинстве случаев непредсказуемы. В то же время пользователь предпочитает, чтобы обслуживание не очень сильно менялось в течение продолжительного времени. В частном случае, вводя программу в машину, пользователь должен иметь основанное на опыте работы с этой программой приблизительное представление, когда ему ожидать выдачи результатов.
5. **Удобства.** ОС должна быть гибкой и удобной для пользования. Очевидно, что это свойство сугубо объективно.

Структура ОС.

Ядро (kernel) – резидентная часть ОС, работающая в режиме супервизора. («обычно» работает в режиме физической адресации).

Интерфейсы системных вызовов
(API – Application Program Interface)
||
Динамически подгружаемые драйверы физических и
виртуальных устройств – обращение
к ним не требует сверх аппарата.
||
Ядро ОС – программы обработки прерываний
и драйверы наиболее быстрых устройств.

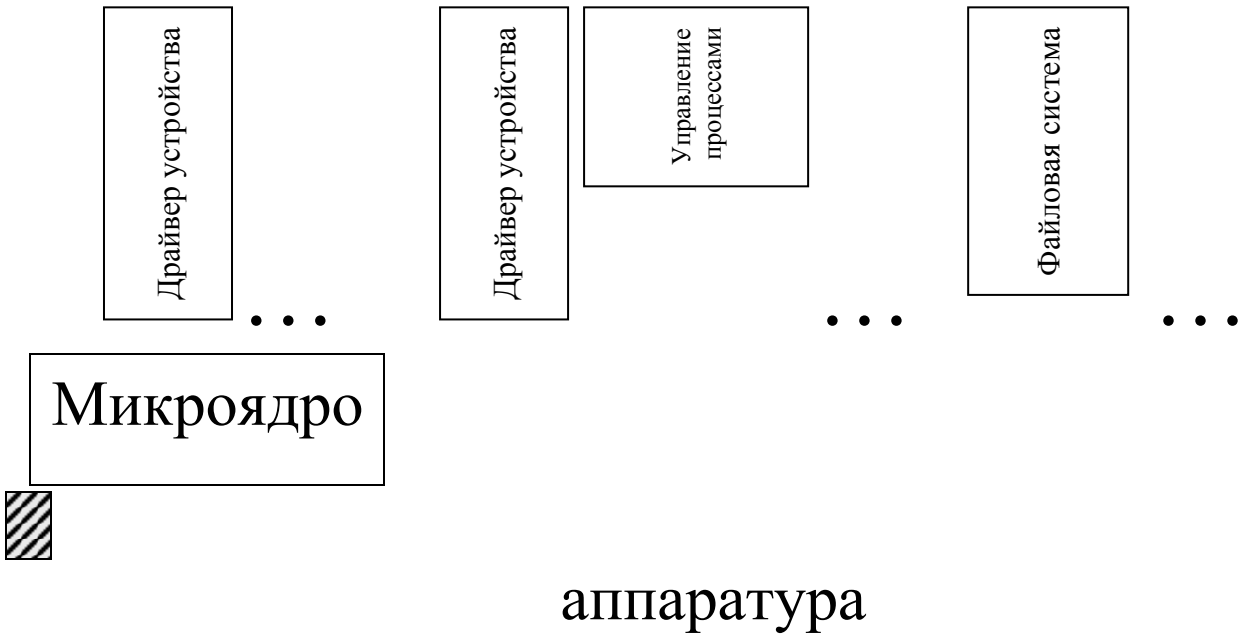
Автоматически включается режим супервизоров.

«Системный вызов» - обращение к ОС за предоставление той или иной функции (возможности, услуги, сервиса).

Описание ⇔ стандартная библиотека программ.

Существует 2 уровня организации ОС:

1) пользовательский, на котором программист пользуется библиотекой через системные вызовы. Реализуется т.н. монолитное ядро – достаточно большой компоновщик, включающий взаимодействие процессоров, драйверы и т.п.
2) микроядерная архитектура, при которой существует микроядро, которое обеспечивает минимальные функции ОС, плюс оно имеет фиксированный набор интерфейсов, которые соотв. Подключению драйверов ОС. Все, что выше микроядра достаточно просто модифицируется.



Логические ф-ции ОС

- управление процессами

суть в функциях ОС, которые обеспечивают образование процесса и его выполнение в рамках ОС.

-

управление ОП

ОС может реализовывать разные стратегии распределения ОП и степени защиты

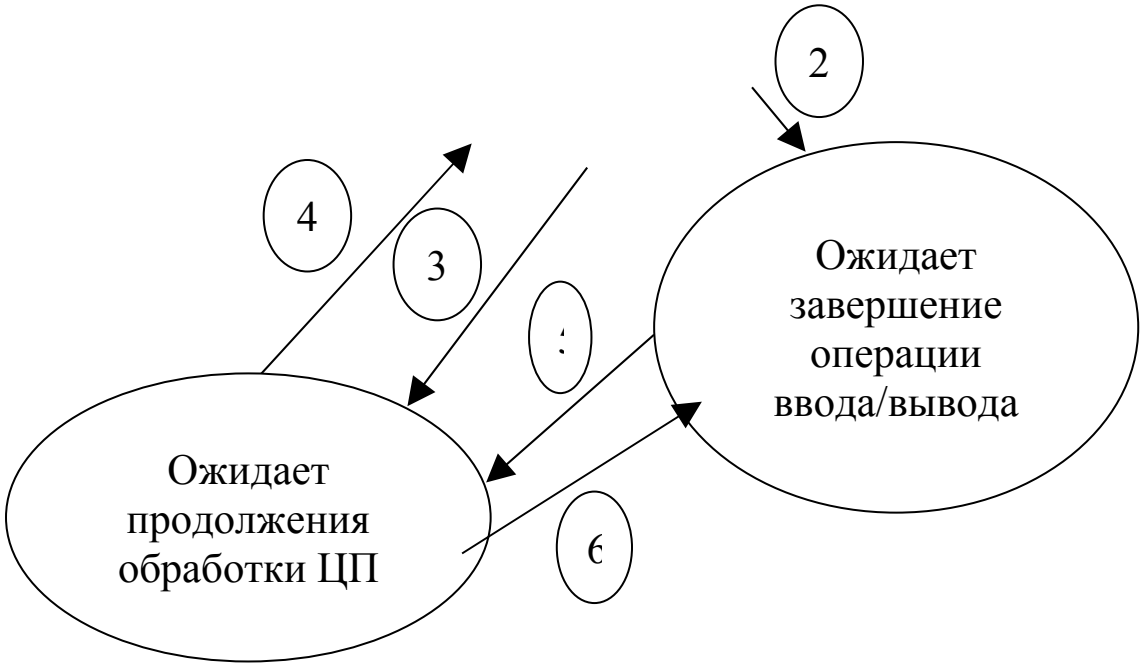
- планирование

выбор очередности программ для начала обработки, распределение энергии для многопроцессорных устройств, доступ к внешним устр-вам

- управление устройствами и ФС

Планирование.
Будем считать, что рассматриваем нек. Модельную ОС, в которой процесс после его формирования поступает на буфер ввода процесса. За время нахождения в буфере формируются данные и структуры для выполнения. Буфер обрабатываемых процессов внешней памяти, в которой находится программа, обрабатывается в мультипрограммном режиме.

Обобщенный жизненный цикл процесса:



Но это только самые явные этапы планирования.

Типы ОС

Многое определяется стратегиями планирования в данных ОС. С точки зрения типов ОС планирования традиционно выделяют 3 типа:

1) Пакетная ОС

Пакет программ **–** некоторая совокупность программ, для выполнения каждой из которых требуется «значительное» время работы процессора.
Первоначально на перфокартах, позже в буфере ввода программ.
Переключение выполнения процессов происходит только в одном из случаев:
●Выполнение процесса завершено
●Возникло прерывание
●Был фиксирован факт зацикливания процесса
Примитивная, но крит. эффект – минимизация накладных расходов.

2) Система разделения времени

Базируется на квантировании времени. **Квант времени ЦП –** некоторый фиксированный ОС промежуток времени работы ЦП.
Переключение выполнения процессов происходит только в одном из случаев:
●Исчерпался выделенный квант времени
●Выполнение процесса завершено
●Возникло прерывание
●Был фиксирован факт зацикливания процесса
варьируя размер кванта можно добиться разных результатов.

3) ОС реального времени

Системы реального времени являются специализированными системами, в которых все функции планирования ориентированы на обработку некоторых событий за время, не превосходящее некоторого предельного значение.
Обычно обеспечивают управление техническим оборудованием.

Примером такой организации планирования ЦП может быть следующая схема. Планировщик построен на двухуровневой схеме. Мы считаем, что множество задач может содержать, предположим, счетные задачи и интерактивные задачи. Первый уровень определяет приоритет между двумя классами задач и либо отдает ЦП сначала счетной задаче, либо интерактивной задаче. А второй уровень определяет то, о чем мы говорили перед этим, т.е. как выбрать задачу в пределах одного класса и как ее прервать. Такая смешанная система может работать следующим образом. Первый уровень планирования будет работать по такому принципу: если в данный момент нет ни одной интерактивной задачи, готовой к выполнению (а это вполне реальная ситуация, если пользователи занимаются редактированием текста), то ЦП передается счетным задачам, но добавляется одно условие: как только появляется хотя бы одна интерактивная задача, счетная задача прерывается и управление передается блоку интерактивных задач.

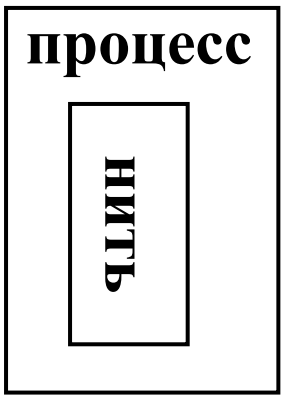
Типы процессов.

«Полновесные процессы» **-** это защищенных участков памяти процессы, выполняющиеся внутри операционной системы, то есть имеющие

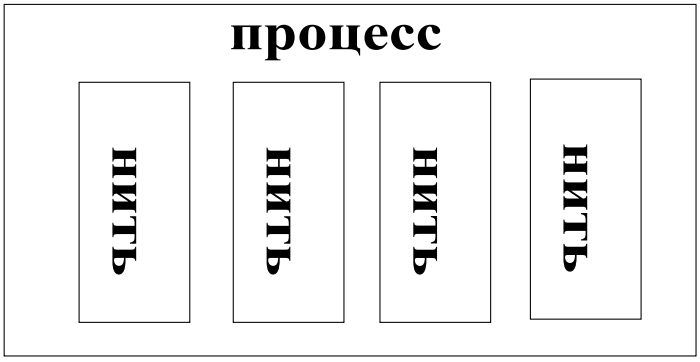
собственные виртуальные адресные пространства для статических и динамических данных.

«***Легковесные процессы» - нити -*** работают в мультипрограммном режиме одновременно с активировавшей их задачей и используют ее виртуальное адресное пространство.

Также процессы можно классифицировать как процессы с однонитевой организацией и с много нитевой.



однонитевой процесс



многонитевой процесс

Понятие процесса можно определить несколькими способами. Как исполняемый код;
собственное адресное пространство, которое представляет собой совокупность виртуальных адресов, которые может использовать процесс;
ресурсы системы, которые назначены процессу ОС;
хотя бы одна выполняемая нить.

Одна из основных характеристик процесса – контекст – совокупность данных, характеризующих актуальное состояние процесса. Контекст включает в себя 2 составляющие:

1) Пользовательская составляющая – текущее состояние программы (совокупность машинных команд, размещенных в ОЗУ)
2) Системная составляющая
   – информация идентификационного характера (PID процесса, PID «родителя»…)
   – информация о содержимом регистров (РОН, индексные регистры, флаги...)
   – информация, необходимая для управления процессом (состояние процесса, приоритет....)

Процессы в ОС UNIX.

Процесс в UNIXe имеет системно-ориентированные характеристики:

- Объект, зарегистрированный в таблице процессов
- Объект, порожденный системным вызовом fork()

Каждому процессу соответствует идентификатор процесса, который идентифицируется с процессом до его завершения.
Контекст процесса в UNIXe – совокупность данных, которые принадлежат адресному пространству процесса и ОС, состоит из 3х компонент:

- пользовательская
- аппаратная
- системная

1)Пользовательская:

Сегмент кода

- машинные компоненты
- системные команды

Сегмент данных

- статические данные
- раздел. Память
- стек

2) Аппаратная составляющая - 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.

3) Системная составляющая:
•приоритет процесса
•реальный и эффективный идентификаторы пользователя-владельца
•текущее состояние процесса
•реальный и эффективный идентификатор группы, к которой принадлежит владелец
•идентификатор родительского процесса

Создание нового процесса.

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

1. Системный вызов fork()

С понятием процесса в ОС UNIX связаны средства формирования процесса, суть их заключается в следующем. Ядром системы поддерживается функция fork(). При обращении к этой функции происходит дублирование процесса. Образуется процесс-двойник, который идентичен процессу-отцу (идентичен, но не совпадает с процессом-отцом!). Функция возвращает значение:

>0, это PID сыновнего процесса (мы находимся в процессе-отце)

=0 (мы находимся в процессе-сыне)

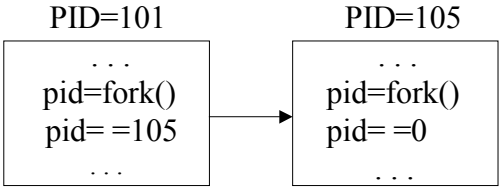
=-1 произошла ошибка - невозможно создать новый процесс, эта ошибка может возникнуть при недостатке места в таблице процессов, при нехватке места в системных областях данных и т.п.

Заметим, во-первых, что fork() - общесистемное средство порождения процессов. Система пользуется этим средством в подавляющем большинстве случаев. Во-вторых, система поддерживает родственные взаимоотношения между процессами, это означает, что существуют некоторые функции, характерные для работы с процессами, которые доступны только процессам, являющимся родственниками. Родственные связи процессов отражаются на использовании тех или иных средств UNIX. При порождении сыновнего процесса с использованием fork() порожденный процесс наследует от «отца»:

- **Окружение** - при формировании процесса ему передается некоторый набор параметров-переменных, используя которые, процесс может взаимодействовать с операционным окружением (интерпретатором команд и т.д.). О переменных окружения мы поговорим на последних лекциях нашего курса;
- **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 действующего идентификатора пользователя** (это то, что связано с s-bit'ом);
- В ОС имеется возможность системными вызовами осуществлять отладку (профилирование) программы, в процессе может быть указано - **разрешено или нет профилирование**;
- **Разделяемые ресурсы** процесса-отца;
- **Текущий рабочий каталог и домашний каталоги**.

Не наследуется при создании нового процесса идентификатор процесса

В качестве иллюстрации работы fork() можно привести следующую картинку:



Здесь процесс с PID=105 создается процессом с PID=101.

Также следует отметить, что если убивается процесс-отец, то новым отцом становится 1-ый процесс ОС.

Пример:

```
int main(int argc, char **argv)
{

     printf("PID=%d; PPID=%d \n", getpid(),
 getppid());
     fork();
     printf("PID=%d; PPID=%d \n", getpid(),
getppid());
}
```

2. Системный вызов exec()

#include <unistd.h>

int execl(const char *path, char *arg0, …, char *argn, 0);

Суть функций exec() - в следующем: при обращении к ней происходит замена тела текущего процесса, в соответствии с именем исполняемого файла, указанного одним из параметров функции. Загружается новый процесс и управление перелается на точку входа. Функция возвращает «-1», если действие не выполнено, и код, отличный от «-1», если операция прошла успешно.

В результате exec() изменяется:

- режим обработки сигналов;
- эффект идентификации владельца;
- файловые дескрипторы.

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  );
     &
}
```

Работа с функциями fork – exec.

Связка fork/exec по своей мощности сильнее, чем, если бы была единая функция, которая сразу бы создавала новый процесс и замещала бы его содержимое. Fork/exec позволяют вставить между ними еще некоторую программу, которая будет содержать какие-то полезные действия. Может возникнуть естественный вопрос: зачем все это нужно? ОС UNIX - конструктор. Система может конструировать сложные объекты за счет функциональных «кирпичиков». Так вот эти кирпичики - это функции fork(), exec(...) и др.

Причинами завершения процесса могут быть:

- системный вызов **_exit()**
- оператора **return**, входящего в состав функции **main()**

- получение некоторых сигналов

Пример:

```
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  ,  echoэто  ,сообщение два  ,NULL);
          printf( ошибка  );
     }

     if(fork() == 0)
     {
          execl(  /bin/echo  ,  echoэто  ,сообщение три  ,NULL);
          printf( ошибка  );
     }
     printf( процесс-предок закончился  )
}
```

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 **status** 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.

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**

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.

Жизненный цикл процесса может быть изображен сдежующим образом:
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.

Последовательность действий при загрузке UNIXa.

1) загрузка ядра системы, а основную память и ее запуск. Наличие системного устройства – устр-ва, на котором аппаратура ищет ОС при старте
   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.

2) инициализация системы включает в себя:
   1. устанавливаются системные часы (для генерации прерываний),

2. формируется диспетчер памяти,

3. формируются значения некоторых структур данных (наборы буферов блоков, буфера индексных дескрипторов) и ряд других.

4. По окончании этих действий происходит инициализация процесса с номером "0".
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 течениее 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.
Сначала процесс "1" представляет собой полную копию процесса "0" , т.е. у него нет области кода. Потом происходит увеличение его размера. 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 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.

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.

Лекция 5. Взаимодействие процессов: синхронизация, тупики

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:

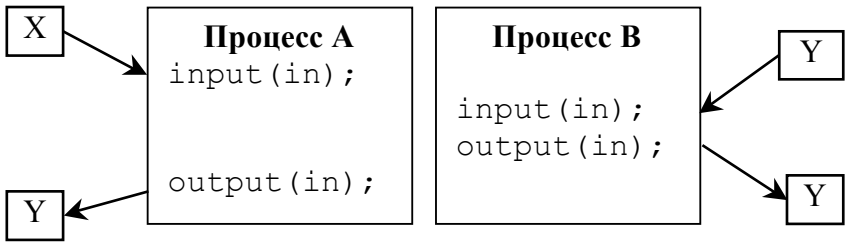


Рис. 1 Конкуренция процессов за ресурс.

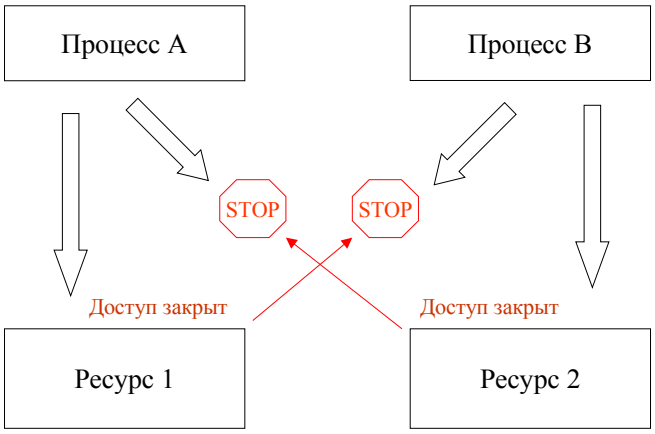
В этом случае символ, считанный процессом **А**, был потерян, а символ, считанный процессом В, был выведен дважды. Результат выполнения процессов здесь зависит от того, в какой момент осуществляется переключение процессов, и от того, какой конкретно процесс будет выбран для выполнения следующим. Такие 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 с 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

Средства синхронизации

Семафоры Дейкстры

Семафоры были предложены как механизм синхронизации доступа к разделяемым ресурсам западноевропейским ученым Дейкстрой (Дийкстра - и такой вариант его имени встречается в литературе), человеком известным в теории программирования (Кстати, именно его учения легли в основу разработок алгоритмов маршрутизации сетей). В литературе эти семафоры часто называют семафорами Дейкстры. Он предложил использовать некий формализм, суть которого заключается в том, что есть некоторая целочисленная переменная S, которая называется семафором. Над семафором определены две операции: P(S), V(S).

- действие P(S) таково: если значение семафора равно нулю, то процесс, выполняющий обращение к P(S) будет приостановлен до тех пор, пока какой-нибудь другой процесс не изменит значение семафора, если же значение семафора отлично от 0, то значение семафора будет уменьшено на единицу.
- V(S) увеличивает значение семафора на единицу;

Считается, что операции P(S) и V(S) неделимы. Это означает, что выполнение этих операций не может прерваться до их завершения. Эти операции состоят как бы из трех команд: чтение переменной S, сравнение с 0 и то или иное действие над S. Так вот эти три команды, три действия неделимы. Никакое событие не может приостановить выполнение этой цепочки действий и передать управление другому процессу. (Прослеживается некоторая аналогия с механизмом прерываний, т. е. если при выполнении команды произошло прерывание, то команда выполнятся до конца, затем происходит обработка прерывания с последующим возвратом в эту же точку программы). Две операции с семафором - как бы две цельные машинные команды. Эти команды должны входить в систему команд или же ОС должна уметь эмулирвать их.

Частным случаем продекларированного семафора является двоичный семафор, максимальное значение которого равно единичке. Если S=1, то это означает, что ни один из процессов (связанных с этим семафором) не находится в критическом участке. При S=0 один из процессов находится в критическом участке {вот-вот попадет в очередь}, а другой нормально функционирует. При S=-1 один из процессов находится в критическом участке, а другой заблокирован и находится в очереди.

На самом деле двоичные семафоры наиболее часто находили применение в аппаратных реализациях, например, в многомашинных комплексах с общей оперативной памятью. Аналогичным образом можно говорить и о т. н. k-ичных семафорах.

Использование двоичного семафора для организации взаимного исключения проиллюстрировано на рисунке.

процесс 1

```
int semaphore;
…
down(semaphore);
/*критическая секция
процесса 1 */
…
up(semaphore);
…
```

процесс 2

```
int semaphore;
…
down(semaphore);
/*критическая секция
процесса 2 */
…
up(semaphore);
…
```

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

Мониторы

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.

Обмен сообщениями

Средство, решающее проблему синхронизации
•для однопроцессорных систем и систем с общей памятью,
•для распределенных систем (когда каждый процессор имеет доступ только к своей памяти)
Основная функциональность метода обеспечивается двумя примитивами (являющимися, как и семафоры, в отличие от мониторов, системными вызовами, а не конструкциями языка) :
send (destination, message)
receive (message)
•Синхронизация
- Операции посылки/приема сообщения могут быть блокирующими и не блокирующими.
• Адресация
-Прямая (ID процесса)
-Косвенная (почтовый ящик, или очередь сообщений)
• Длина сообщения

Классические задачи синхронизации процессов

«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]);
     }
}
```

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.

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.

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**.

```
#define CHAIRS 5
typedef int semaphore;  /* тип данных «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();          /* процесс стрижки */
      }
      else
      {
        up(mutex);              /* нет свободного кресла для ожидания –
                                    придется уйти */
      }
}
```

Лекция 6 .Основы взаимодействия сети.

Время однопроцессорных компьютеров уходит. Даже те, которые мы называем однопроцессорными, по сути, многопроцессорные. Так, у них могут быть дополнительные функции обработки информации (видео, вз-е с внешними устройствами…) Но, с др стороны, приходит время , когда понимание процессора, как обработчика программы, тоже уходит => необходимость появления многопроцессорных архитектур. Причины:

- информационное общество (необх-ть получения инф-ции извне – Internet);
- появление спектра задач, для решения кот.невозможно применять стандартные методы однопроцессорных систем, даже из-за объема этих задач
- необх-ть исп-я параллельных арххитектур компьютеров.

Многомашинные и многопроцессорные ассоциации.

С точки зрения арх. Компьютеров существует много подходов к классификации. У всех есть свои достоинства и недостатки.

Классификация архитектур Майкла Флина (M. Flynn). Суть состоит в том, что рассматриваются 2 потока : поток инструкций (команд), и поток данных. Вот основные комбинации этих потоков:

• **ОКОД** (**SISD** – single instruction, single data stream) – традиционный, имеется 1 компьютер с единственным ЦП, на который подаются одиночные порции команд.

• **ОКМД** (**SIMD** - single instruction, multiple data stream) – матричная обработка данных. На 1 устройство управления поступают множественные потоки данных, т.е. для каждой инструкции можно выбрать порцию данных.

• **МКОД** (MISD - multiple instruction, single data stream) – в некотором смысле вырожденная каткгория, хотя к ней относятся, например, специальные параллельные графические системы.

• **МКМД** (**MIMD -** multiple instruction, multiple data stream) – множество процессоров одновременно выполняют различные последовательности команд над своими данными. Так или иначе 2 и более процессора со своими устройствами управления, каждое из кот. Может выполнять свою программу. Конвейерность становится свойством компа, а МКМД на сегодняшний день наиболее актуальны.

МКМД в свою очередь подразделяются на:

1) Системы с общей оперативной памятью – для всех процессорных элементов общая ОП, используемая программа берется из единого места, к которому разрешен доступ от любого процесса.

- UMA –система с однородным доступом в память (uniform memory access), например, SMP – симметричная мультипроцессорная система (symmetric multiprocessor)

Удобны тем, что a'priory настроены не параллелизм, недостаток - Существуют ограничения на количество процессорных элементов

. Проблема с организацией КЭШа: либо КЭШ отдельный в кажом процессе элемента, либо создать общий КЭШ (но это уже КЭШ 2ого уровня);

- NUMA – системы с неоднородным доступом в память (nonuniform memory access)

. У каждого процесса своя локальная память и нелокальная. Доступ к локальной быстрее. Проще решаются проблемы с кэшированием, т.к. Степень параллелизма выше чем SMP.

2) Системы с распределенной оперативной памятью. Каждый элемент обладает уникальной памятью. Эта категория похожа на специализированную компьютерную сеть. Суть : имеется компьютерные элементы (процессоры с локальной ОП) и средства коммуникации, которые их объединяют.

- COW – (Claster of workstations –
кластер рабочих станций). Самые популярные на сегодняшний день. Это многопроцессорные системы, которые объединены специальной быстрой сетью и предназначены для решения задач. (ex, TOP100, TOP500). Использование традиционных комплектующих и средств коммуникации для создания высокоэффективных систем с возможностью наращивания мощности
системы. Проблема охлаждения.

- MPP - (Massively
Parallel Processors) –
процессоры с массовым
параллелизмом. промышленное развитие кластера. Использование спец ср-в коммуникации под конкретные условия. Они более универсальны.

Другая категория многомашинных ассоциаций – **терминальные комплексы**. Они предназначены для организации массового доступа удаленных и локальных пользователей к ресурсам некоторой вычислительной системы. Их используют для

- сбора и централизации обработки информации;
- обеспечения массового доступа к ресурсам вычислительных систем.

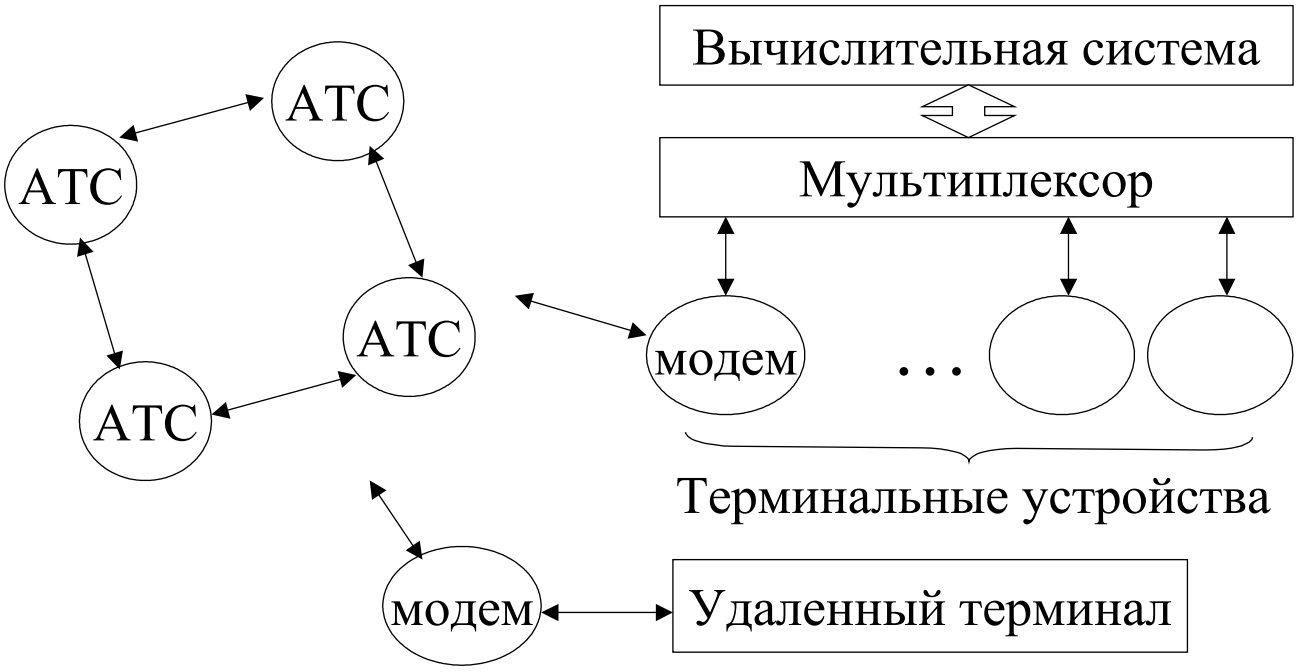
Терминальный комплекс может включать в свой состав:

•основную вычислительную систему (массовый доступ локальных и удаленных пользователей)

•локальные мультиплексоры (подключения к выч сист через 1 или неск каналов ввода \ вывода группы внешних устройств)

•локальные терминалы \ терминальные устройства (подкл непосредственно к компу через мультиплексор. Это могут быть устройства, которые обеспечивают вз-е пользователя с вычислительной системой или др пользователями. Предполагает исп-е модемов)

•удаленные терминалы (имеющие доступ к ресурсам выч системы через телефонные виды связи, модемы или удаленные процессоры)



<u>проблемы коммуникации.</u>

Для взаимодействия, как правило, телефонную сеть, которая предполагает какого либо количества телефонных связей и выходов на устройства.
Существует 2 разновидности каналов:

• Коммутируемый канал, суть которог заключается в том, что когда мы поднимаем трубку, мы даем на ближайшую АТС инф-цию о том, что говорим с абонентом. Кол-во одновременно обрабатываемых линий меньше, чем подключенных. 2 проблемы : время на коммутацию и возможность отказа в соединении.

• Выделенный канал – связь на постоянной основе, но стоимость существенно выше, хотя гарантируется качество соединения.

Линия связи одного терминала с комплексом называется Канал точка-точка.
Терминальный комплекс с группой удаленных компьютеров называется Многоточечный канал.

С точки зрения организации потоков различают:

•Симплексные каналы – обмен только в 1 направлении

•Дуплексные каналы – совмещены потоки в обе стороны (телефон)

•Полудуплексные каналы – может и в двух направлениях, но в каждый определенный момент только в одном (рация)

Компьютерные сети

Другая составляющая, которая определяет необходимомть развития компьютерных систем. С точки зрания архитектуры, она очень похожа, если заменить терминальные устройства на компьютеры. В общем случае, Компьютерная сеть – объединение компьютеров (или вычислительных систем), взаимодействующих через коммуникационную среду. **Коммуникационная среда** – каналы и средства передачи данных

Свойства компьютерной сети:

- состоит из достаточного числа взаимодействующих между собой компов, обеспечивающих сбор, хранение и обработку информации.
- Предполагает т.н. распределенную обработку инф-ции, т.е. может обрабатываться на разных узлах, а результат – комплекс
- Расширяемость сети как по протяженности, так и по пропускной способности каналов связи и производительности компов4
- Возможность использования симметричных интерфейсов.

Абонентские или основные компьютеры называются хосты, остальные коммуникационные или вспомогательные компьютеры (шлюзы, маршрутизаторы, ...)

Компьютерные сети исторически подразделяются на 3 категории:

• Сеть коммутации каналов. Обеспечивает установление канала связи на время всего сеанса связи между абонентскими машинами. Хорошо: если канал коммутирован - нет никаких накладных расходов (со скоростью самого медленного звена канала). Плохо: сеанс связи может быть произвольной длительности => возможна деградация сети. Без гарантированной продуктивности.

• Сеть коммутации сообщений. Сеанс связи представляется в виде последовательности сообщений, т.е. порций данных определенного размера. При запуске сообщения коммутация соединения не устанавливается, а сообщение отправляется в сеть с соответствующей информацией о маршруте. Машина считает сообщение и отправит дальше. Если свободных каналов нет, то временно сообщение будет сохранено на коммуникационной машине. Проблемы: должна быть инф-ция о порядке сообщений; несмотря на повышение дискретности передачи канала, размер сообщения может быть неопределенным. достоинства: убрали промежутки «молчания» между сообщениями; не устанавливается канал – экономится канальный ресурс.

• Сеть коммутации пакетов. Все сообщения разделяются на блоки ланных некоторого фиксированного размера, который не превосходит некоторую предельную границу. Работает также, как коммутации сообщений, но в пакетном режиме.

Реальные сети строятся на комбинации этих 3х категорий.

Организация сетевого взаимодействия

Одна из осн проблем - проблема стандартизации. Суть в том, что изначально – корпоративные локальные сети => перенос программ из одной сети в др крайне тсложен. Опыт развития некоторого прообраза сети Internet.
ISO разраотало систему открытых интерфейсив (OSI) от уровня передачи сигналов (физ вз-я в сети) до прикладного вз-я (доступ к сетевым услугам).
Снизу вверх – семиуровневая система ISO \ OSI. Каждый уровень – условный набор действий для передачи информации в сети выч систем (аппаратная или программная составляющая)
Протокол связи – формальное описание сообщений и правил, по которым сетевые устройства (вычислительные системы) осуществляют обмен информацией. Или Правила взаимодействия одноименных уровней.

В каждом уровне ISO \ OSI предполагается наличие некоторого кол-ва протоколов, кажд из которыз может осуществлять работу с такими же на др. машине (в т.ч. виртуальной)

1. Физический уровень. Однозначно определяется физ среда передачи данных и форматы передачи сигналов (оптоволокно, изорнет, витая пара…)
2. Канальный уровень. Обеспеч управление доступом к физ среде передачи данных, в частности, синхронизация передачи данных, формализация правил передачи данных, задачи обнаружения и локализации ошибок
3. Сетевой уровень. В нек смысле ключевой, т.к. решается вопрос управления связью между вз-щими компами, маршрутизация.
4. Транспортный уровень. Уровень логического канала. Проблема управления и передачи данных, обеспечение порядка.
5. Сеансовый уровень. Упр-е сеансами связи. Проблема обеспечения синхронизации отправки данных, защита прав и нештатные ситуации.
6. .Представительский уровень. Проблема унификации кодировок.
7. Прикладной уровень. Стандартизация вз-я с прикладными системами.
8.

Интерфейс – правила взаимодействия вышестоящего уровня с нижестоящим.
Служба или сервис – набор операций, предоставляемых нижестоящим уровнем вышестоящему.
Стек протоколов – перечень разно уровневых протоколов, реализованных в системе

Семейство протоколов TCP / IP

Служит основой для организации Interneta.
1. Уровень доступа к сети
2. Межсетевой уровень
3. Транспортный уровень.

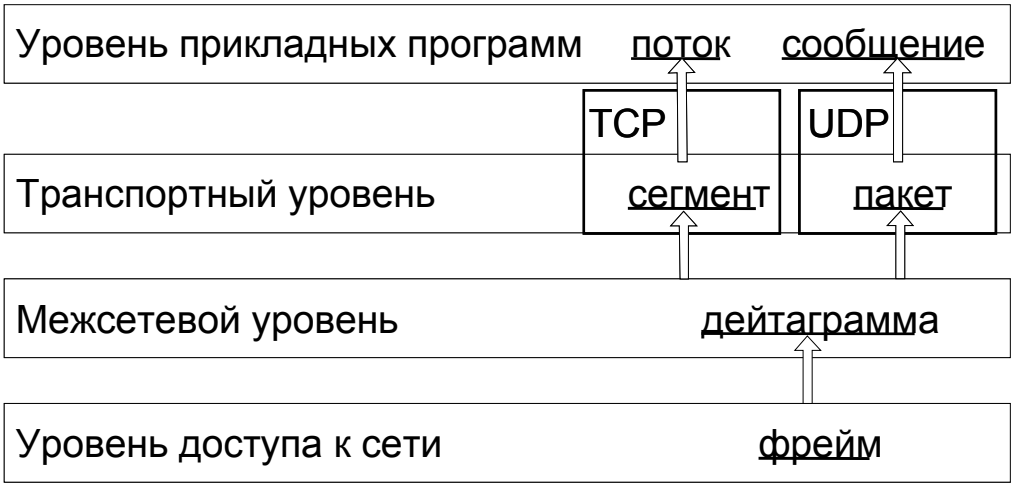
4. Уровень прикладных программ.

Свойства:

- открытые стандарты прграмм, кот поддерживаются почти всеми операционными средами;
- могут работать с использованием всех имеющихся средств передачи данных;
- обладают уникальной системой именования сетевых устройств;
- стандартизованные протоколы прикладных программ.

В соответствие с уровнями ISO \ OSI:

Уровень модели TCP/IP	Уровень модели ISO/OSI
1. Уровень доступа к сети.	Канальный уровень Физический уровень
2. Межсетевой уровень	Сетевой уровень
3. Транспортный уровень	Сеансовый уровень Транспортный уровень
4. Уровень прикладных программ	Уровень прикладных программ Уровень представления данных



Рассмотрим эти уровни подробнее:

1. Уровень доступа к сети. На этом уровне протоколы обеспечивают систему средствами для передачи данных другим устройствам в сети

2. Межсетевой уровень В отличие от сетевого уровня модели OSI, не устанавливает соединений с другими машинами.

•Функции протокола IP

- формирование дейтаграмм (Дейтаграмма – это пакет протокола IP.

Пакет – блок данных, который передаётся вместе с информацией, необходимой для его корректной доставки.)

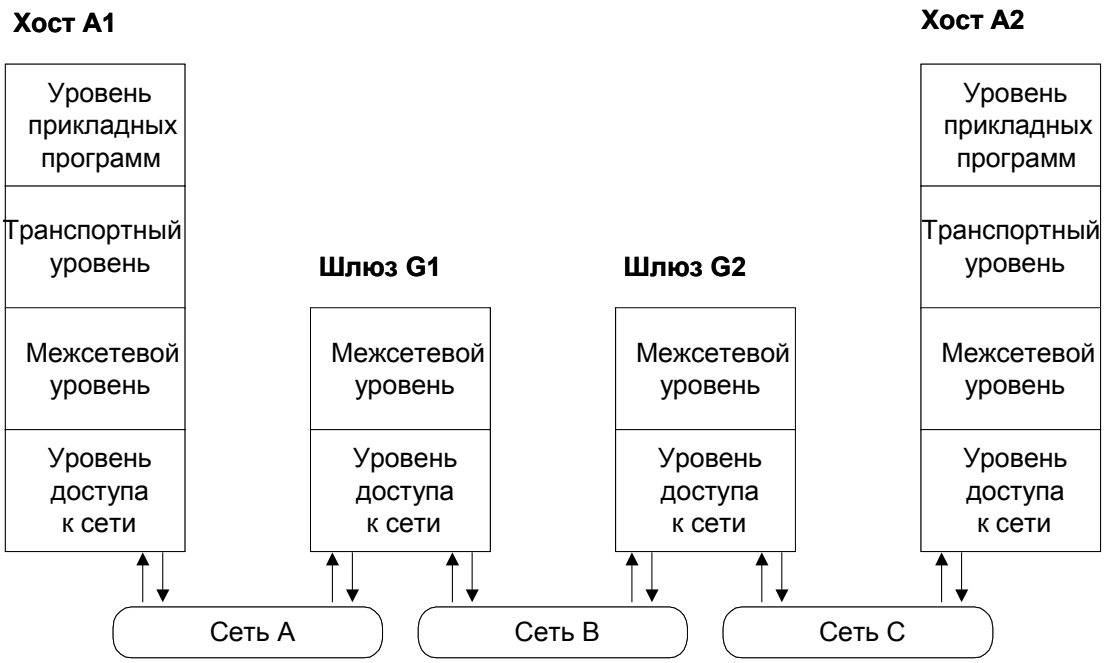
- поддержание системы адресации
- обмен данными между транспортным уровнем и уровнем доступа к сети
- организация маршрутизации дейтаграмм (Маршрутизация – процесс выбора шлюза или маршрутизатора.

Шлюз – устройство, передающее пакеты между различными сетями)

- разбиение и обратная сборка дейтаграмм
•IP – протокол без логического установления соединения
•Протокол IP не обеспечивает обнаружение и исправление ошибок

система адресации протокола **TCP / IP**

3. Транспортный уровень. Обеспечивает доставку данных от компьютера к компьютеру, обеспечивает 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, UDP.
•Протокол контроля передачи (TCP, Transmission Control Protocol) - обеспечивает надежную доставку данных с обнаружением и исправлением ошибок и с установлением логического соединения.
•Протокол пользовательских дейтаграмм (UDP, User Datagram Protocol) - отправляет пакеты с данными, «не заботясь» об их доставке.



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.

.•Протоколы, опирающиеся на TCP

•TELNET (Network Terminal Protocol)

•FTP (File Transfer Protocol)

•SMTP (Simple Mail Transfer Protocol)

•Протоколы, опирающиеся на UDP

•DNS (Domain Name Service)

•RIP (Routing Information Protocol)

•NFS (Network File System)

Лекция 7. Файловые системы

Многие из многопроцессорных вычислительных компьютеров строятся как сети. Если рассматривать кластер, как объединение компьютеров, и локальную сеть, то, в принципе, мы увидим, что это практически одно и то же, с точностью до оборудования. Разница состоит в том, что сетевые ОС обеспечивают функционирование распределенных приложений, а кластер – это некий единый компьютер с более простой сутью, т.н. распределенной ОС. Между компьютерами, образующими кластер, распределяются нек. Ф-ции ОС, такие как:

- управление процессами, планирование, организация и синхронизация
- функции файловой системы

Эти компьютеры, в основном, работают в пакетном режиме. Их задача – минимизирование времени работы ОС.

Файловые системы.

Одним из важных компонентов операционной системы является средство управления данными операционной системы, которое обычно называют файловой системой. Итак, файловая система - это компонент операционной системы, обеспечивающий организацию создания, хранения и доступа к именованным наборам данных. Эти именованные наборы данных называются файлами.

Еще недавно доступ к данным производился только через адреса носителей, а координатами данных были имя устройства, номер блока и список остальных => организация данных зависела от того, как использовался этот носитель. Это было неудобно, во-первых, потому, что надо помнить о местоположении файла, а, во-вторых, чтобы перенести файл в другое место, надо было написать программу.

Появления ФС как систем именованного доступа совершило революцию. Появилась возможность ассоциировать совокупности данных некоторое имя. Появились ссылки на координаты относительно начала. ОС брала на себя обязанности по сохранению и защите данных.

С точки зрения структуры организации данных существует ряд подходов:

1. последовательность байтов (без структуры);
2. последовательность записей fix размера (исторически появ из-за использования перфокарт. читалось по 80 байтов.)
3. файлы, имеющие организацию записей переменной длины (либо имеющие спец инф-цию о длине, либо со спец маркером конца.) Исключался прямой доступ к записям. Изменение длины существующей записи приводило к определенным проблемам.)
4. иерархическая организация файла (структура в виде дерева. В кажд узле находится информация о записи: поле ключа и поле данных. Место расположения записи в общем случае произвольно.)

с точки зрения прагматики, в современном мире многие ФС – это комбинации данных типов.

С точки зрения ФС, каждый файл имеет набор параметров, которые определяют свойства этого файл, т.н. атрибуты файла:

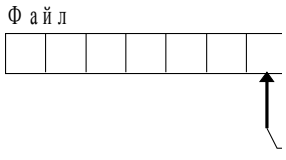
- имя
- права доступа к файлу
- персонификация (владелец, пользователи)
- тип файла (правила интерпретации содержимого файла)
- размер записи (не обяз)

- указатель чтения \ записи (все операции доступа относятся к некоторой позиции, на которой находимся)
- другое

= время последующей модификации
= время последнего обращения
= …

Практически каждая 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. А операционная система, исходя из этой информации, может принять какие-либо решения (например, блокировать, разрешить или синхронизировать доступ к этому файлу со стороны других процессов).
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) Создать новый файл. Функция создает новый файл. В некоторых ОС создание файла осуществляется по функции открытия файла.
4) Чтение/запись. Обычно обмен с файлами может организовываться некоторыми блоками данных. С одной стороны, размеры этих блоков данных могут варьироваться программистом, с другой стороны реальные физические ресурсы также могут иметь блочную структура и, следовательно, определенный размер блока. Получается, что эффективность обменов, а, следовательно, и эффективность работы всей ВС в целом, в данном случае зависит от умения программиста, потому что именно он при использовании функций чтения/записи определяет длину блоков, которыми будет осуществляться обмен. Например, размер блока нашего жесткого диска равен 512 Кб, а мы читаем довольно приличный объем данных «порциями» по 128 Кб, то это значительно отразится на эффективности работы нашей программы, не смотря на то, что некоторые «умные» операционные системы пытаются сглаживать эти элементы неэффективности.



5) Управление файловым указателем. Практически с каждым открытым файлом связывается т. н. файловый указатель. Этот указатель, по аналогии с регистром счетчика команд (а скорее даже по аналогии с регистром-указателем на вершину стека), 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 управления работы с файлами требуется уметь управлять этим указателем. В операционных системах имеется функция, позволяющая произвольным образом перемещать указатель в пределах файла. Доступ к содержимому файла может быть, по аналогии со способами работы с ВЗУ, прямым (двигаем указатель куда

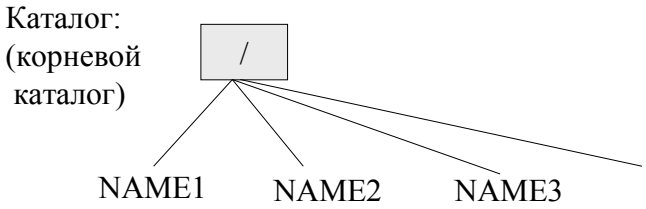
нам нужно в пределах файла и читаем / пишем) и последовательным (чтобы прочесть i-ый блок данных необходимо сначала прочесть первые i-1 блоки). Вообще говоря, файловый указатель есть некоторая переменная, доступная программе, которая создается при открытии файла.

Типовые программные интерфейсы работы с файлами

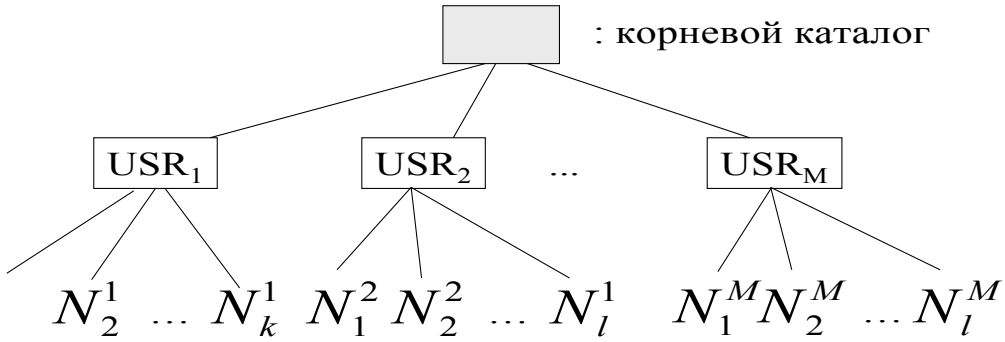
open	– открытие / создание файла
close	– закрытие
read / write	– читать, писать (относительно положения указателя чтения / запись)
delete	– удалить файл из файловой системы
seek	– позиционирование указателя чтение/запись
rename	– переименование файла
read / write _attributes	– чтение, модификация атрибутов файла.

Один из важнейших элементов ФС – организация каталога – части ФС, содержащей инф-цию о содержащемся в ФС файле. Нек ФС рассматривают каталог как файл спец типа (UNIX). Но если это файл, то можно использовать стандартные интерфейсы для работы с файлом.
С точки зрения организации каталога, могут быть

1) одноуровневая модель (в ФС один каталог, где перечислены все файлы, находящиеся в ФС) проблемы:
- коллизия имен
- при большом объеме – определенная нагрузка при работе
- неудобно структурировать организацию файла

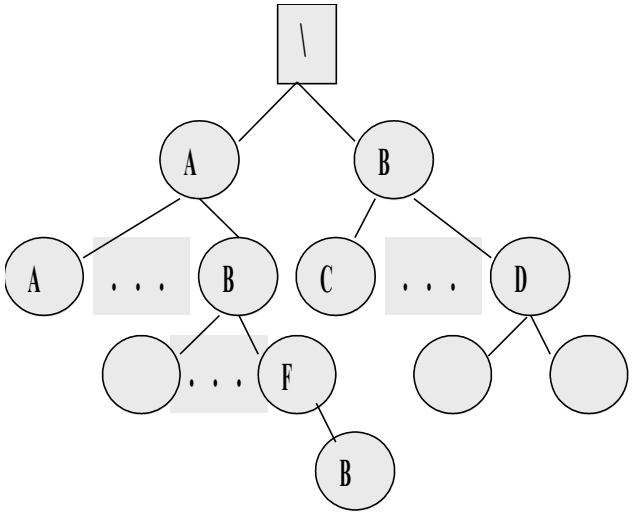


2) двухуровневая (совокупность каталогов, которые могут быть ассоциированы с пользователями. Для каждого пользователя – одноуровневая структура.) Остается проблема структурирования.



файлы пользователей

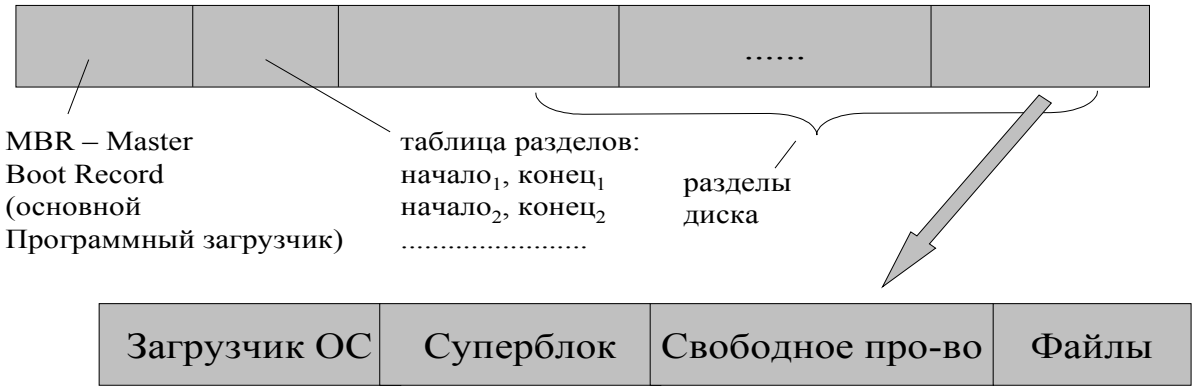
3) Иерархическая (древообразная) (листья – файлы или каталоги, в узлах, где нет листьев – каталоги. Каждый файл имеет уникальный путь до корня, и он может быть представлен в виде перехода. Этот путь называется <u>полным именем файла</u>.



Каждый файл, кроме корня, зарегистрирован в каком-либо каталоге. Его <u>имя</u> – имя в соответствующем каталоге. <u>Относительное имя</u> – имя в текущем каталоге. Т.о. у каждого файла 3 имени.

<u>Практическая реализация ФС.</u>

Системный загрузчик – нулевой блок системного устройства.



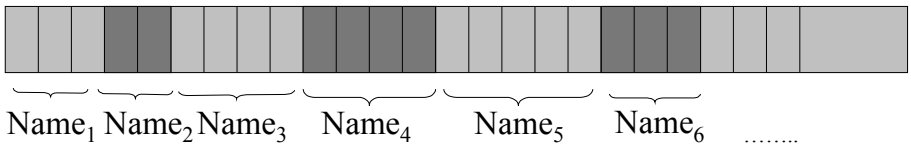
Загрузчик ОС	Суперблок	Свободное про-во	Файлы
знает всю инф-цию о данной ОС	инф-ция о настройках ФС и об активном состоянии		

В общем случае блок как таковой м.б. представлен в системе:

1) Блок устройства «HDD» (дискового \ ориентированного устройства) – определяется св-вами этого устр-ва;
2) Блок файловой системы – первичный уровень виртуальности, настройки ФС;
3) Блок файла – можно определить размер блоков для работы с файлами.

Модели реализации файла.

1) Непрерывные файлы (только последовательные блоки ФС)



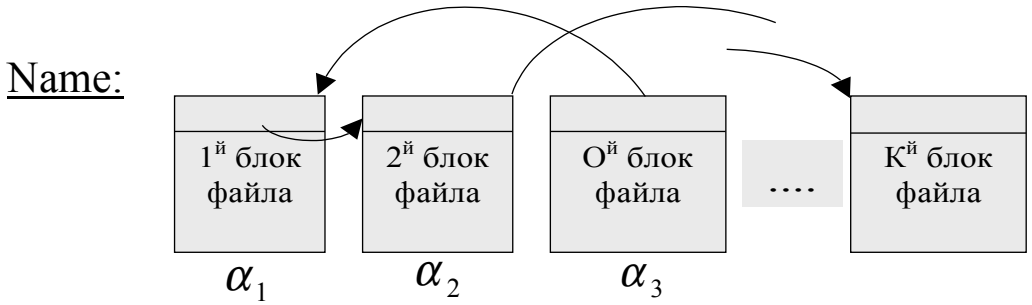
Достоинства:

- Простота реализации;
- Эффективность по доступу;
- Высокая производительность

Недостатки:

- Фрагментация свободного пространства (прямая и косвенная);
- Проблема внутренней фрагментации;
- Увеличение размера существующего файла.

2) Файлы, имеющие списочную структуру (в нулевом блоке ссылка на первый и т.д.)



$\{ \alpha_i \}$ - - множество блоков файловой системы в которых размещены блоки файла Name

Достоинства:

- Отсутствие фрагментации свободного пространства (за исключением блочной блочной фрагментации)
- Простота реализации
- Эффективный последовательный доступ

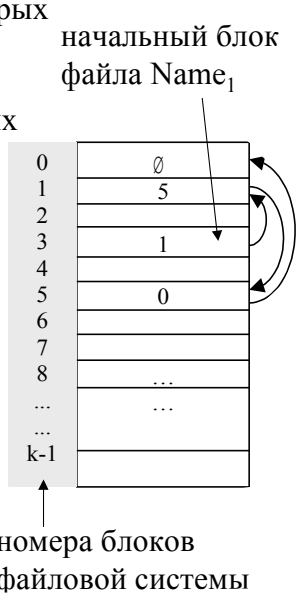
Недостатки:

- Сложность (не эффективность) организации прямого доступа

- Фрагментация файла по диску
- Наличие ссылки в блоке файла (ситуации чтения 2-х блоков при необходимости чтения данных объемом один блок).

3) Таблицы размещения (DOS)

Таблица, в которой кол-во строк соответствует кол-ву блоков, в которых размещены блоки ФС.



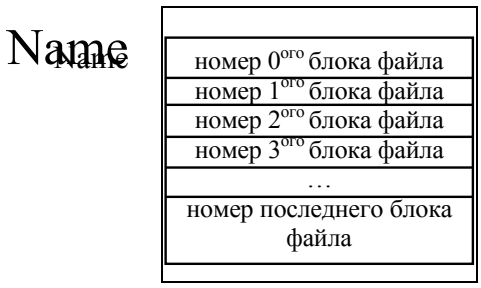
Достоинства:

- возможность использования всего блока для хранения данных файла;
- оптимизация прямого доступа (при полном или частичном размещении таблицы в ОЗУ).

Недостатки:

- желательно размещение всей таблицы в ОЗУ (проблема размера, например для 60 Gb раздела и блоков размером 1Kb потребуется 60 000 000*4b = 240 Mb).

4) Индексные узлы (дескрипторы) - системная структура данных, содержащая информацию о размещении блоков



конкретного файла в файловой системе.

Достоинства:

- нет необходимости в размещении в ОЗУ информации всей FAT о все файлах системы, в памяти размещаются атрибуты, связанные только с открытыми файлами.

Недостатки:

- размер файла и размер индексного узла (в общем случае прийти к размерам таблицы размещения). Решение:
  - ограничение размера файла
  - иерархическая организация индексных узлов

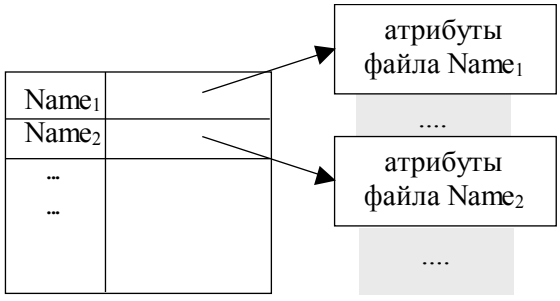
-

Модели организации каталогов

$Name_1$	Атрибуты
$Name_2$	Атрибуты
…	
…	

Простейший каталог: Записи каталога фиксированного размера, содержат имя файла и все его атрибуты.

Каталог содержит имя файла и ссылку на

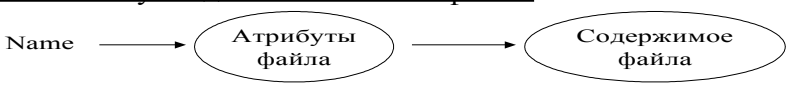


системную структуру данных, в которой размещены атрибуты файла. Размер атрибутов может
варьироваться.

Взаимно однозначное соответствие между именем файла и его содержимым.

1. Содержимому любого файла соответствует единственное имя файла.

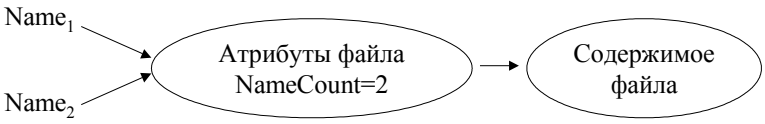
Например,



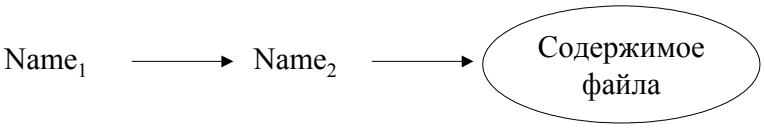
Или древообразная иерархическая файловая система

2. Содержимому файла может соответствовать два и более имен файла.

«Жесткая связь»



«Символьные связи»



Пространство внешней памяти

Координация использования пространства внешней памяти.

1. Определение оптимального размера блока ФС:

- если «Большой блок»:
  - эффективность обмена
  - существенная внутренняя фрагментация

  (не эффективное использование пространства ВП)
- если «Маленький блок»:
  - эффективное использование пространства ВП
  - фрагментация данных файла по диску

Развитые системы позволяют параметризовать эту характеристику.

2. Учет свободных блоков ФС:

- исп-ние связного списка свободных блоков. ФС изначально выделяет в пространстве раздела блоки, предназначенные для хранения ссылок на следующие блоки. В ОП - содержимое или часть содержимого 1ого блока, если израсходовалось – читается след блок и т.д.
- исп-ние битового массива. Все пространство представляется ввиде битовой карты.

Состояние любого блока определяется содержимым бита с номером каждого
блока.

Если блок свободен, бит равен 1, занят – 0.

3. Квотирование пространства ФС:

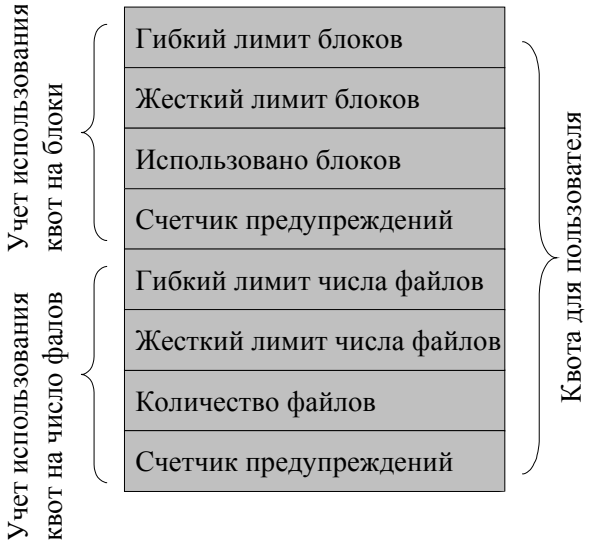
Это Учет количества файлов и их размеров у которых атрибут владельца соответствует конкретному пользователю.

2 категории ресурсов:

- количество имен файлов (ограничивается размером раздела)

- пространство, которое занимают эти файлы

Жесткие лимиты не превышаются никогда. Гибкие квоты можно превышать, но после этого «включается обратный счетчик» предупреждений. Пока счетчик != 0, при каждой регистрации пользователя в системе он получает предупреждение; если счетчик = 0, пользователь блокируется.



Надежность ФС

При работе с ФС возможна потеря информации в результате аппаратного или программного сбоя или случайное удаление файлов. Есть несколько вариантов решения этой проблемы:

1)Резервное копирование в общем случае должно происходить в ограниченные промежутки времени.

•Копируются не все файлы файловой системы (избирательность архивирования по типам файлов);
• Инкрементное архивирование (резервное копирование) – единожды создается «полная» копия, все последующие включают только обновленные файлы;
• Использование компрессии при архивировании (риск потери всего архива из-за ошибки в чтении/записи сжатых данных);
• Проблема архивирования «на ходу» (во время копирования происходят изменения файлов, создание, удаление каталогов и т.д.)
• Распределенное хранение резервних копий.

2)Архивирование
Стратегии:
•Физическая архивация
- «один в один» (забывая о структуре ФС);
- интеллектуальная физическая архивация (копируются только использованные блоки файловой системы);
- проблема обработки дефектных блоков.
•
Логическая архивация – копирование файлов (а не блоков), модифицированных после заданной даты.

3)Проверка целостности или непротиворечивости файловой системы.

При аппаратных или программных сбоях возможна потеря информации:

- потеря модифицированных данных в «обычных» файлах;
- потеря системной информации (содержимое каталогов, списков системных блоков, индексные узлы и т.д.)

Уже на ходу, в режиме профилактики

1.Формируются две таблицы:

- таблица занятых блоков;
- таблица свободных блоков;

(размеры таблиц соответствуют размеру файловой системы – число записей равно числу блоков ФС)

Изначально все записи таблиц обнуляются.

2. Анализируется список свободных блоков. Для каждого номера свободного блока увеличивается на 1 соответствующая ему запись в таблице свободных.

3. Анализируются все индексные узлы. Для каждого блока, встретившегося в индексном узле, увеличивается его счетчик на 1 в таблице занятых блоков.

4. Анализ содержимого таблиц и коррекция ситуаций.

Возможно 4 случая:

1.

0	1	2	3	4	5	
1	1	0	1	0	1	Таблица занятых блоков.
0	0	1	0	1	0	Таблица свободных блоков.

Непротиворечивость файловой системы соблюдена.

2.

0	1	2	3	4	5	
1	0	1	0	1	1	Таблица занятых блоков
0	0	0	1	0	0	Таблица свободных блоков.

В таблице обнаружился пропавший блок : - Можно оставить как есть, но система «замусоривается»;

- Добавить в список свободных блоков файловой системы.

3.

0	1	2	3	4	5	
1	0	0	1	0	1	Таблица занятых блоков.
0	1	2	0	1	0	Таблица свободных блоков.

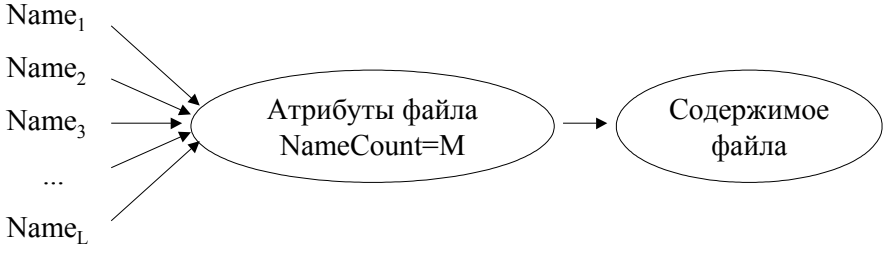
Образуется дубликат свободного блока – пересоздание списка свободных блоков.

Дубликат занятого блока =>

автоматическое решение

максимально затруднено, имеет место потеря информации в одном из файлов.

5. Проверка непротиворечивости файлов ФС:



Возможные варианты:

1. L== M – все в порядке

2. L != M - ошибка

Лекция 8. ОС UNIX. Файловая система.

UNIX – революционная ФС, потому что это –

1. первый системный программный продукт, разработанный с использованием языка высокого уровня (обычно – assembler или макрооператоры к нему);
2. элегантная и развитая система управления процессами (fork-exec);
3. особенности ФС:
   - архитектура ФС. UNIX использует древовидную организацию ФС, которая иногда превращается в ориентированный граф, более сложный, чем дерево.
   - исп-ние концепции файлов. Практически все в UNIX-машине представлено в виде файлов. Все через единый интерфейс (раньше все сист вызовы были отдельные).
4. В системе достаточно аккуратно и прагматично организована работа с внешними устройствами. Одно и то же устройство можно в UNIX определить и как байт - и как блок-ориентированное.
5. ОС UNIX получила широкое распространение благодаря «прозрачности» принимаемых системой решений, простоте организации системных данных, алгоритмов и взаимосвязей.
6. UNIX - «переносимая» операционная система. Это означает, что большая часть кода, алгоритмов легко переносятся на другие архитектуры.

Виды файлов в ОС UNIX

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

Виды файлов:

• обычный файл (regular file). Данные, кот ввели и кот получаем после их выполнения.

• каталог (directory)

• специальный файл устройств (special device file). Файлы, имеющие спец тип, посредством кот можно использовать те или иные драйверы устройств в системе UNIX.

• именованный канал (named pipe). (FIFO) регулярные файлы, имеющие определенную фикс систему доступа

• ссылка (link). Спец файлы, кот могут нарушать древовидную организацию.

• сокет (socket). Спец файлы, предназначенные для вз-я процессов как в рамках одной локальной машины, так и в рамках сети.

Права доступа

Классическая модель рассматривает 3 категории пользователей:

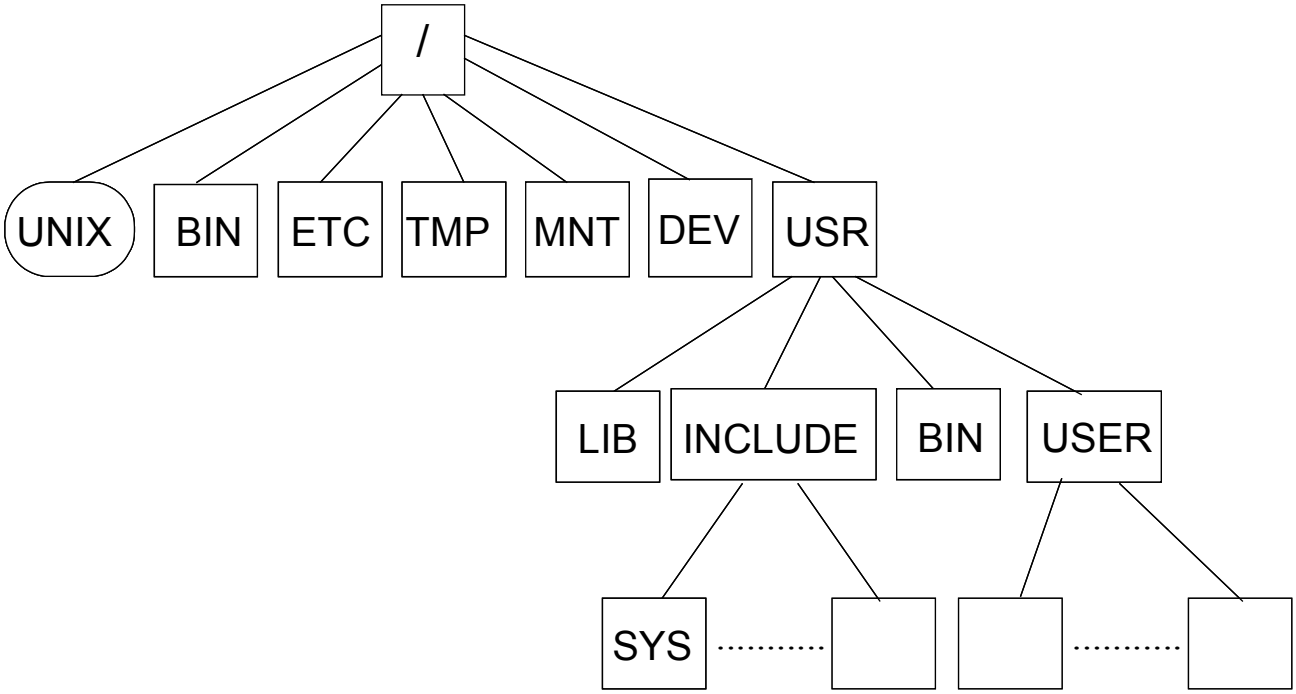
- пользователь (владелец)
- группа
- все пользователи системы

Права каждой категории:

- чтение
- модификация
- запуск на исполнение

права интерпретируются в зависимости от типа файла.

Логическая структура ФС



UNIX предлагает некоторую стандартную структуру ФС, a'priory предполагающую определенные имена каталогов с файлами.

Есть корневой каталог **UNIX** с точностью до именования. Он содержит программные ядра ОС или часть ядра.

BIN **каталог с исполняемыми файлами наиболее распространенных команд UNIXa. Можно перестроить команды в нек каталогах или сделать их разными для разных пользователей.**

ETC **файл, в кот хранится системная информация, обеспечивающая разного рода настройки в системе, в т.ч. passwd (инф-ция о всех зарегистрированных в системе пользователях).**

TMP - **временные файлы ФС, сохранность которых не гарантируется после перезапуска системы.**

MNT - **корневой каталог локальной ФС ассоциируется с к-л каталогом в MNT.**

DEV - **спец файлы устройств.**

USR - **размещается пользовательская инф-ция.**

LIB – **например gcc, cc.**

include **содержит headerы, кот используются программой пользователя для препроцессора. Например,** SYS **содержит include – файлы системы.**

BIN **содержит исполняемые файлы, которые предоставляются для пользовательского доступа и характеризуют конкретную установку. (так, если знаем имя файла, сначала ищем его в USR/BIN, а затем в BIN.**

USER – **домашни каталоги зарегистрированных пользователей.**

Внутренняя организация ФС

1. **модель версии SYSTEM V**

Суть: ФС в дисковом разделе имеет 3 категории пространства:

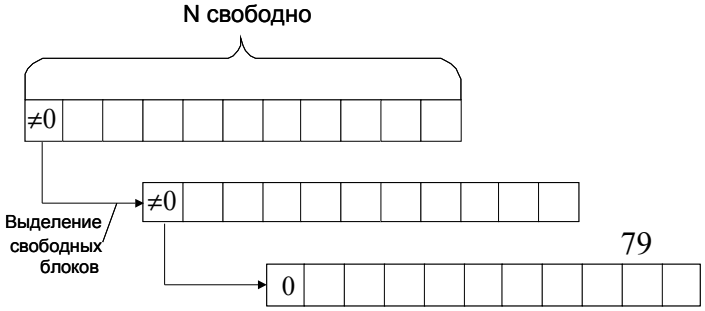
- спецобласть – часть ФС, в кот находится инф-ция о настройках ФС и актуальном состоянии ФС;
- одласть индексных дескрипторов – системная структура данных, которая описывает состояние файла в ФС;
- область рабочих блоков ФС, в кот системные структуры данных, блоки файлов и содержимое файлов.

Суперблок файловой системы содержит оперативную информацию о текущем состоянии операционной системы, а также данные о параметрах настройки файловой системы. В частности, суперблок содержит информацию о количестве, так называемых, индексных дескрипторов в файловой системе. Также суперблок содержит информацию о количестве блоков, составляющих файловую систему, а также информацию о свободных блоках файлов, о свободных индексных дескрипторах, и прочие данные, характеризующие время, дату модификации и другие специальные параметры.

За суперблоком следует 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. Каждый индексный дескриптор содержит следующую информацию:

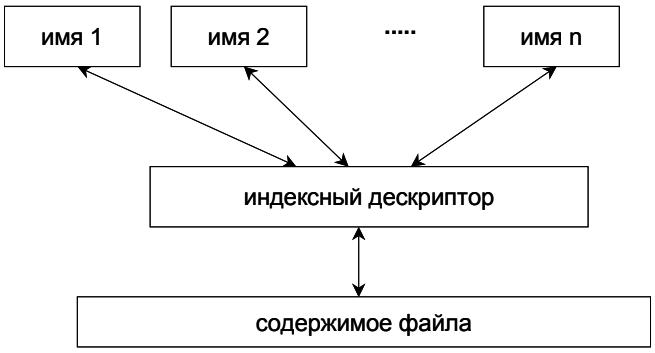
1. Поле, определяющее тип файла (каталог или нет).
2. Поле кода защиты (доступ).
3. Количество ссылок к данному индексному дескриптору из всевозможных каталогов файловой системы (в ситуации нарушения дерева файловой системы). Если значение этого поля равно нулю, то считается, что этот индексный дескриптор свободен.
4. Длина файла в байтах.
5. Статистика: поля, характеризующие дату и время создания и т.п.
6. Поле адресации блоков файлов.
7. массив номеров блоков файла.

Массив номеров свободных блоков файла. В суперблоке: все свободные блоки ФС организованы в однонаправленный список: 1й эл-т списка – массив из N ссылок, кот размещены в суперблоке. Нулевой эл-т - № блока из пространства блоков ФС, в котпрод-е списка. ФС



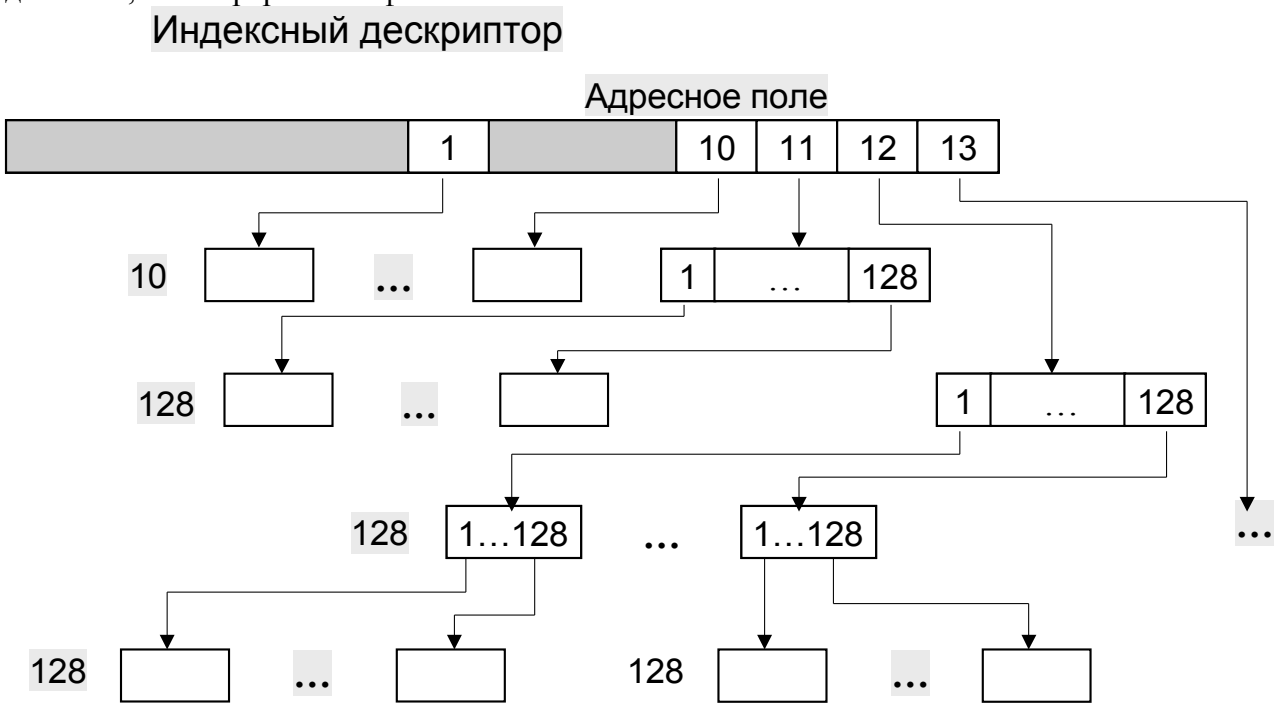
работает оперативно с этим массивом. Если много освобождаем – выбираем след блок и складываем.

Массив свободных индексных дескрипторов. Массив фиксированного размера, без списочной организации. Запускается спец процесс:



•Освобождение ИД
- Есть свободное место – номер –> элемент массива
- Нет свободного места – номер "забывается"
•Запрос ИД
- Поиск в массиве
  - Массив пустой – обновление массива
  - Массив не пустой - ok

Массив блоков файлов, кот находятся в ИД. В общем случае состоит из 13ти эдементов. Первые 10 – имена 10ти логич блоков памяти. К ним доступ осуществляется без к-л накладных расходов. Если больше 10ти – включается в работу косвенность: 1 уровень – 11й элемент – 128 номеровблоков проджолжения памяти. Еслим мало – слово №12, в нем ссылка на № блока, в кот 128 номеров блоков, где каждый содержит 128 № блоков памяти и т.д. Лучше всего может длинный, но непрерывный файл.



Файл - каталог

Мы с вами говорили, что одним из свойств операционной системы UNIX является то, что вся информация размещается в файлах, т.е. нет каких-то специальных таблиц, которыми пользуется операционная система, за исключением тех таблиц, которые она создает, уже функционируя в пространстве оперативной памяти. Каталог, с точки зрения файловой системы, - это файл, в котором размещены данные о тех файлах, которые принадлежат каталогу.

В каталоге A содержаться файлы B, C и D, несмотря на то, что файлы B и C могут быть как файлами, так каталогами, а файл D является каталогом.

Каталог состоит из элементов, которые содержат два поля. Первое поле - номер индексного дескриптора, второе поле - это имя файла, которое ассоциировано с данным индексным дескриптором. Номера индексных дескрипторов (в пространстве индексных дескрипторов) начинаются с единицы. Первый индексный дескриптор - индексный дескриптор каталога. В общем случае в каталоге могут встречаться записи, ссылающиеся на один и тот же индексный дескриптор, но в каталоге не могут быть записи, имеющие одинаковые имена. Имя в пределах каталога уникально, но с содержимым файла может ассоциироваться произвольное количество имен. Поэтому есть некоторая неоднозначность в определении понятия файл в операционной системе UNIX. Файл оказывается не просто именованным набором данных: у него есть индексный дескриптор и может быть несколько имен (т.е. имя - вторичная компонента).

При создании каталога в нем всегда создаются две записи: запись на специальный файл с именем «.» (точка), с которым ассоциирован индексный дескриптор самого каталога, и файл «**..**» (две точки), с которым ассоциируется индексный дескриптор (ИД) родительского каталога. Для нашего примера каталог A имеет, например, ИД с номером 7, а каталог D имеет ИД с номером 5. Файл F имеет ИД №10, файл G имеет ИД №101. В этом случае файл-каталог D будет иметь следующее содержимое:

Имя	№ИД
«**.**»	5
«**..**»	7
«F»	10
«G»	101

Первая запись - запись на самого себя.
Вторая запись - на родителя (каталог A).
Далее перечислены файлы, которые находятся в этом каталоге.
Вот таким будет содержимое каталога D.

Отличие файла-каталога от обычных файлов пользователя заключается в содержимом поля типа файла в ИД. Для корневого каталога поле родителя будет ссылаться на него самого.

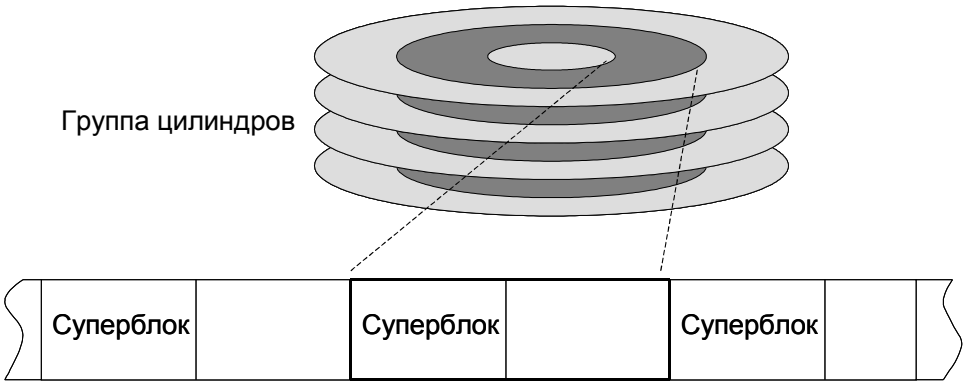
Достоинства версии SYSTEM V:
•Оптимизация в работе со списками номеров свободных индексных дескрипторов и блоков.
•Организация косвенной адресации блоков файлов

Недостатки:
•Концентрация важной информации в суперблоке
•Проблема надежности
•Фрагментация файла по диску
•Ограничения на возможную длину имени файла

ФС UNIX имеет недревовидную систему органтизации, т.к. появился аппарат ссылок. 2 возможности ссылок:

- т.н. «жесткая ссылка» - с одним и тем же идентификатором ассоциируется 2 и более имени, размещенных в произвольных точках ФС, всен они рабноценны;
- «текстовая» (символьная) – спец файл-ссылка, сод кот размещается в идентификаторе и содержимое – текстовая строка, указывающая полное имя файла, с кот ассоциируется новое имя.

2. **Альтернатива для SYSTEM V – FFS BSD.**



Содержится:

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

Характеристики:

1) кластеризация данных;
2) оптимизация физ размещений;
3) оптимизация блоков 2х уровней;
4) позволяет работать с достаточно большими размерами каталогов, но меняет их структуру.

Структура каталога FFS

Фрагментация каталога
Прямой поиск
→ Дефрагментация
Кэширование имен файлов

Управление внешними устройствами.

Управление внешними ресурсами рассматривается как функция, но в основном базируется на аппаратных возможностях ОС. Это скорее компонент ВС, а не ОС, т.к. ОС – это программа с некоторыми свойствами.

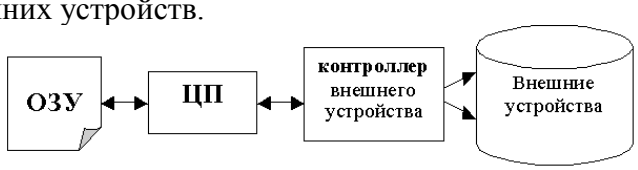
Архитектура организации процессора и внешних устройств.

1. **Непосредственное управление** внешними устройствами центральным процессором.

историческая модель, основанная на том, что ВСЕ действия по управлению внешними устройствами осуществляется через ЦП. Через эти устр-ва можно передавать инфтрмацию о статусе устр-ва (ошибка) и т.п.
предполагается наличие двух потоков:
- потока управляющей инф-ции
- потока данных

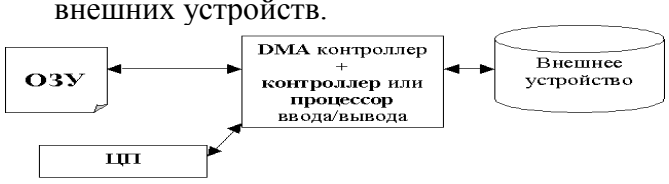
непостедственное управление х-ся тем, что оба потока обслуживаются ЦП.
=> необходим достаточно большой объем личной работы.

2. **Синхронное** управление внешними устройствами с использованием контроллеров внешних устройств.



Суть контроллера – упрощение работы ЦП, хотя оно все равно непостредственное; функции обнаружения ошибок; функции, которые обеспечивают ЦП несколько более высокоуровневый интерфейс к внешним процессам (например. Позиционирование головки, выход на нужный сектор).

3. **Асинхронное** управление внешними устройствами с использованием контроллеров внешних устройств.



Х-ся появлением аппарата прерываний, но оба потока в ЦП.

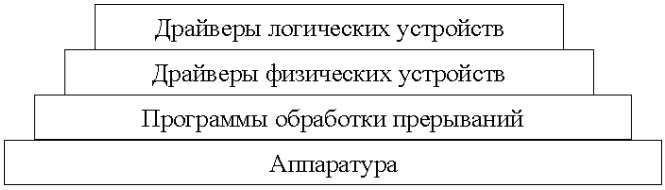
4. Использование **контроллера прямого доступа** к памяти (DMA) при обмене.

ЦП подает управляющую инф-цию по организации обмена, но поток образующихся данных идет между устройствами и ОП, без ЦП.

5. Управление внешними устройствами **с использованием процессора или канала ввода/вывода**.

Обеспечивается высокоуровневый интерфейс для организации ввода\вывода.

Лекция 9. Программное управление внешними устройствами



Цели:

•унификация программных интерфейсов доступа к внешним устройствам (унификация именования, абстрагирование от свойств конкретных устройств);

•обеспечение конкретной модели синхронизации при выполнении обмена (синхронный, асинхронный обмен);

•обработка возникающих ошибок ;

•буферизация обмена (кэширование потоков инф-ции);

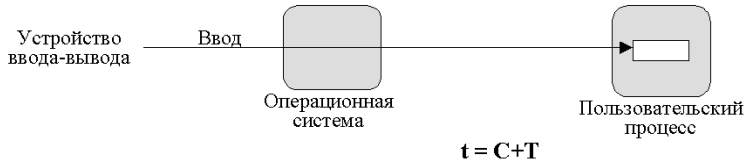
•обеспечение стратегии доступа к устройству (например, многопроцессорное +принтер);

•планирование выполнения операций обмена.

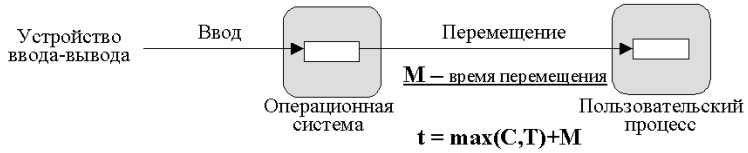
Буферизация обмена

T – время обмена;
C – время выполнения программы между обменами
t – общее время выполнения программы
Схемы буферизации ввода-вывода

а) Без буферизации

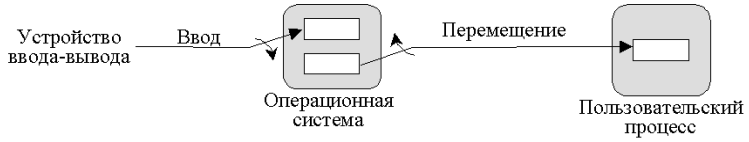


б) Одинарная буферизация



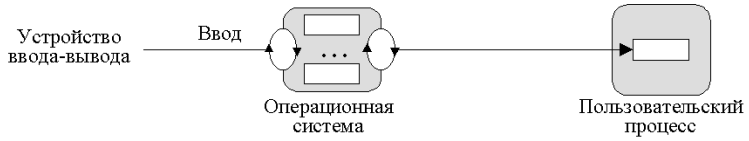
когда буфер освобождается, читается следующий буфер, т.к. считалось то, что потребуется

в) Двойная буферизация



в один буфер — данные для обмена, другой – за предыдущий обмен.

г) Циклическая буферизация



по циклическому списку буферов

Какую схему выбрать зависит от интенсивности буферизации и особенности действий.

Планирование дисковых обменов

поток заказов на обмен превосходит возможности системы в некоторые моменты времени:

1) минимизация
2) разделение приоритетов
3) случайний выход – очередь и датчик случайных чисел

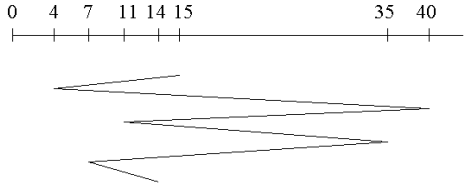
Рассмотрим модельную ситуацию:
головка HDD позиционирована на дорожке 15
Очередь запросов к дорожкам: 4, 40, 11, 35, 7, 14

Варианты решения

1. простейшая модель – случайная выборка из очереди

2. FIFO

Путь головки	L
15 → 4	11
4 → 40	36
40 → 11	29
11 → 35	24
35 → 7	28
7 → 14	7
общ. 135 средн. 22,5	



3. SSTF

Shortest Service Time First – «жадный» алгоритм – на каждом шаге поиск обмена с минимальным перемещением

Путь головки	L
15 → 14	1
14 → 11	3
11 → 7	4
7 → 4	3
4 → 35	31
35 → 40	5
общ. 47 средн. 7,83	

4.

эффективно когда один потребитель читает примерно в одном месте

LIFO

Путь головки	L
15 → 14	1
14 → 7	7
7 → 35	28
35 → 11	24
11 → 40	29
40 → 4	36
общ. 126 средн. 20,83	

5. PRI – алгоритм, основанный на приоритетах процессов.
Проблема – голодание (для всех кроме FIFO), запрос может «зависать» из-за прихода наиболее приоритетных.

SCAN

Путь головки	L
15 → 35	20
35 → 40	5
40 → 14	26
14 → 11	3
11 → 7	4
7 → 4	3
общ. 61 средн. 10,16	

6.

Алгоритм «лифта» - сначала «движение» в одну сторону до «упора», затем в другую, также до «упора». Для ∀ набора запросов перемещений ≤ 2-х число дорожек

C-SCAN

Путь головки	L
15 → 4	11
4 → 7	3
7 → 11	4
11 → 14	3
14 → 35	21
35 → 40	5
общ. 47 средн. 7,83	

7.

Циклическое сканирование
Сканирование в одном направлении. Ищем минимальный номер дорожки, затем «движемся наверх»

8. N-step-SCAN
Разделение очереди на подочереди длины ≤ N запросов каждая (из соображений FIFO). Последовательная обработка очередей. Обрабатываемая очередь не обновляется. Обновление очередей, отличных от обрабатываемой.

RAID системы.

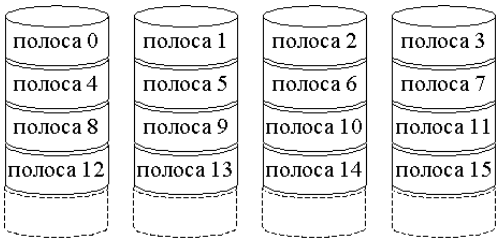
RAID – Redundant Array of Independent (Inexpensive) Disks – избыточный массив независимых (недорогих) дисков. Сначало было не iindependent, a inexpensive.
RAID система - набор физических дисковых устройств, рассматриваемых операционной системой, как единое дисковое устройство (данные распределяются по физическим устройствам, образуется избыточная информация, используемая для контроля и восстановления информации).
Достигается повышение скорости и надежности.

Проблемы

1) **эффективность (когда есть все уровни КЭШ, но нет интенсивности потоков)**
2) **одно из основных качеств программного решения – надежность (24 часа 7 дней в неделю)**

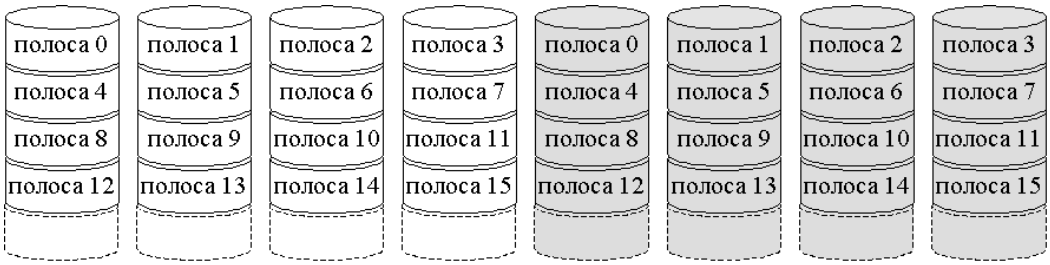
Семь уровней RAID систем.

RAID 0 (без избыточности)



совокупность независимых друг от друга устр-в. нет избыточности инфориъмации. Есть возможность предельного доступа в пределах 1 цикла. Полосы, попавшие на ракзные дисковые усир-ва могут быть параллельными. Ускорение дисковых систем.

RAID 1 (зеркалирование)



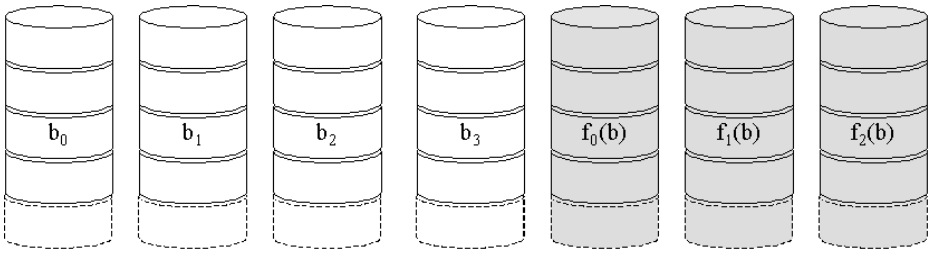
наличие массива дисковых устройств, которые делятся на 2 группы:

- 1 – циклическое распределение полос нулевого уровня
- 2 – копия первой

т.о. мы имеем полную копию входных данных.

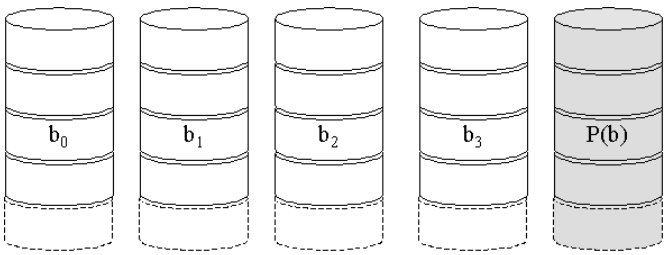
уровни 2 и 3 – уровни синхронизации RAID-мвссивов

RAID 2 избыточность с кодами Хэмминга (Hamming, исправляет одинарные и выявляет двойные ошибки)



часть дисковых устр-в для хранения содержат информацию логического диска. Схема кодов Хэмминга с исправоением одинарных и опрделением двойныхт ошибок.

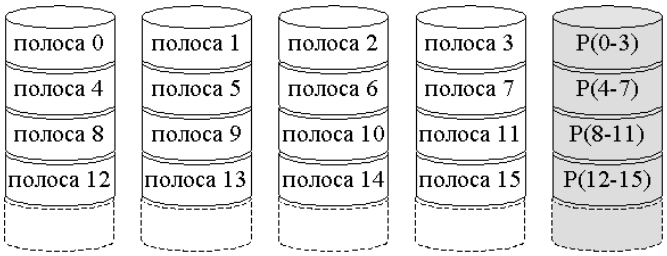
RAID 3 (четность с чередующимися битами)



5 дисков, в них 4 содержательные, а 1 контрольная избыточная информация (XOR для всех 4х).

Пример: 4 диска данных, один – четности:
Потеря данных на первом диске
X4(i)=X3(i)XOR X2(i)XOR X1(i)XOR X0(i)
X1(i)=X4(i)XOR X3(i)XOR X2(i)XOR X0(i)

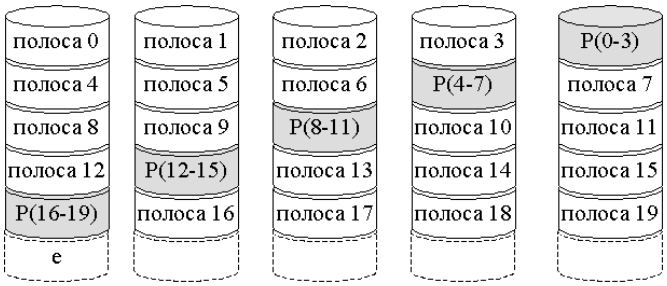
RAID 4



не синхронизированный. Схема примерно та же.

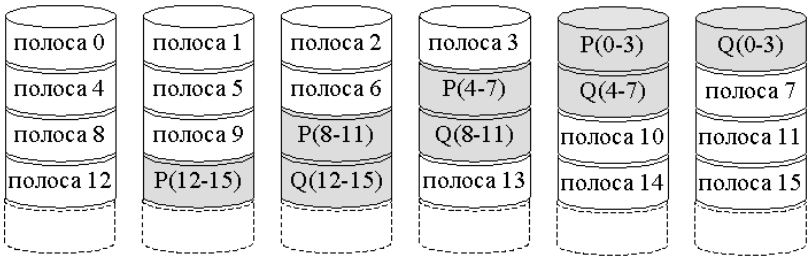
Пример: 4 диска данных, один – четности:
Изначально для любого бита:
X4(i)=X3(i)XOR X2(i)XOR X1(i)XOR X0(i)
После обновления полосы на диске X1:
X4new(i)=X4(i)XOR X1(i)XOR X1new(i)

RAID 5 (распределенная четность – циклическое распределение «четности»)



циклическое распределение контрольного диска

RAID 6 (двойная избыточность – циклическое распределение четности с использованием двух схем контроля: N+2 дисков)



ОС Unix: Работа с внешними устройствами

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.

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;
- **βintr()** – обработка прерывания, вызывается ядром при возникновении прерывания в устройстве с которым ассоциирован драйвер;

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.).

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

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

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.

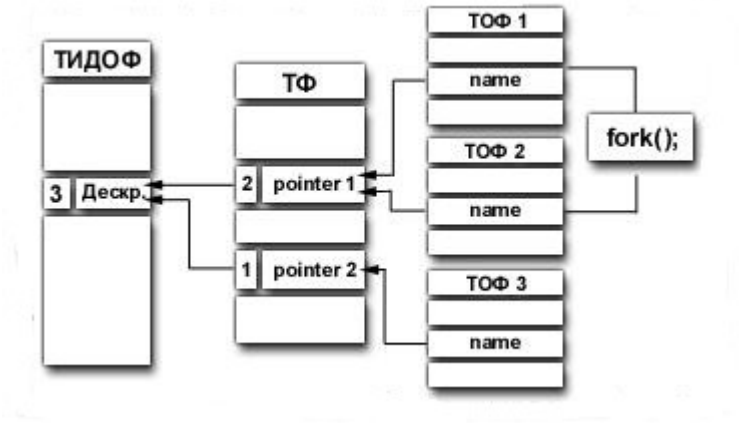
Таблица файлов.

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 прародител

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

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.

Буферизация при блокориентированном обмене

Особенностью работы с блок ориентированными 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.

Лекция 10. Система межпроцессного взаимодействия IPC

По правам доступа можно выделить следующие виды вз-я:

- Симметричное, когда все процессы имеют одинаковые права (напр., каналы или сигналы);
- Асимметричное, когда одному процессу предоставляется набор прав, отличных от предоставленных другому (трассировка).

По тому, кто может участвовать во взаимодействии:

- в пределах одной локальной системы;
- в пределах компьютерной сети.

Понятно, сто в сети взаимодействующие процессы именуются по IP адресам, но как их именовать в пределах локальной машины?

- использовать идентификатор процесса, но это накладывает ряд ограничений:

1) надо знать этот идентификатор (а он связывается с конкретным процессом лишь в момент его появления в системе);
2) недетерминированность (номера случайны);
3) все вз-щие процессы должны участвовать в процессе;

- вз-е родственных процессов. Св-ва, связанные с взаимодействием передаются за счет наследования (неименованные каналы). Исчезает недетерминированность, но могут вз-вать с жесткими ограничениями (должны быть родственниками);
- именованные каналы (можно обращаться как к файлу в произвольное машинное время).

В определенном смысле, система IPC изначально появилась как альтернатива к именованным каналам, как средство организации работы в произвольные моменты времени. Для доступа к именованному каналу надо знать имя файла. Есть некоторый ресурс, в общем случае произвольный, и к этому ресурсу могут добираться все, кто может именовать этот ресурс. Для именования такого рода ресурсов в системе предусмотрен механизм генерации т. н. ключей. Суть его заключается в следующем. По некоторым общеизвестным данным (это могут быть текстовые строки или цифровые комбинации) в системе генерируется уникальный ключ, который ассоциируется с разделяемым ресурсом. Если процесс подтверждает этот ключ, и созданный разделяемый ресурс доступен для него, то после этого он может работать с указанным разделяемым ресурсом по своему усмотрению.

Выделяют следующие разделяемые ресурсы:

1) «очередь сообщений»;
2) разделяемая память;
3) Семафоры.
4)

Каждый IPC ресурс обладает набором атрибутов, среди них:

- атрибут владельца ресурса (эффект идентификатора процесса, создавшего ресурс);
- права доступа к нему (запись \ чтение).

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.

Как сгенерировать такой уникальный ключ?

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)
```

где filename –имя существующего в ФС файла и его атрибутыж

proj – некоторая буква, которая позволяет генерировать разные значения ключа по одному и тому же значению первого параметра – строки.

Эта функция генерирует значение ключа по некоторой строке символов и добавочному символу (букве), 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. Если же файл удалить из ФС, а затем создать новый с тем же именем, вызвать ftok() с тем же параметром, то гарантированно создастся другое имя, т.к. например, учитывается файловый дескриптор.

Следует заметить, что функция ftok() не является системным вызовом, а предоставляется библиотекой.

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.

Ели указано значение **IPC_PRIVATE**, 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 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:

– 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**

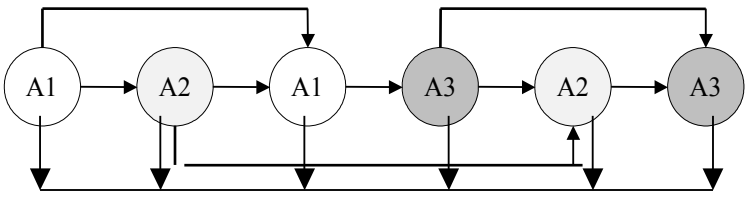
Нужно заметить, что для каждого типа объектов IPC существует некое ограничение на максимально возможное количество одновременно существующих в системе объектов данного типа. Если при попытке создания нового объекта окажется, что указанное ограничение превышено, «**get»-**метод, совершавший попытку создания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

«Очередь сообщений»

Очередь сообщений – организованная совокупность типизированных сообщений, организованное по принципу FIFO. Имеет 2 поля:

- тип сообщеня;
- тело сообщения 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 некоторая совокупность 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.

1) Доступ к очереди сообщений.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
int msgget (key_t key, int msgflag)
```

где **key** – ключ

msgflag – флаги, управляющие поведением вызова

В случае успеха вызов возвращает положительный дескриптор очереди, который может в дальнейшем использоваться для операций с ней, в случае неудачи -1.

2)

Отправка сообщения.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg)
```

где msqid – идентификатор очереди, полученный в результате вызова msgget()

msgp – указатель на буфер следующей структуры:

long msgtype -тип сообщения

char msgtext[ ] -данные (тело сообщения)

msgsz - предельный размер сообщения (буфера)**.**

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.

msgflg - 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.

3)

Получение сообщения.

```
#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)
```

где -

Первые три аргумента аналогичны аргументам предыдущего вызова

msgtyp 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.

msgflg - 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.

4)
Управление очередью сообщений.

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

int msgctl (int msqid, int cmd, struct msgid_ds *buf)
где msqid -
идентификатор ресурса,
cmd - команда, которую необходимо выполнить. Возможные значения:
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.
***buf -** 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.

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с очередью и уничтожения очереди.

Рассмотрим пример.
Напишем программу: первый процесс будет читать 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 и затем будет осуществлен выход. Процессы A и B распечатывают полученные строки на стандартный вывод.

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()
{
key_t key;  int msgid;  char str[256];
key=ftok("/usr/mash",'s');
// получаем 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,  посылаем сообщение в очередь
                msgsnd(msgid,   (struct   msgbuf*)   (&Message), strlen(str)+1, 0);
            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);
            exit(0);
    default: exit(0);
    }
  }
}
```

Процесс-приемник А

```
// процесс  В 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()
{
```

```
key_t key;  int msgid;
key=ftok("/usr/mash",'s'); // получаем ключ по тем же параметрам
msgid=msgget(key, 0666 | IPC_CREAT); //создаем очередь сообщений
for(;;)
  {                          // запускаем вечный цикл
  msgrcv(msgid, (struct msgbuf*) (&Message), 256, 1, 0);
  // читаем сообщение с типом
  if (Message.Data[0]='q' || Message.Data[0]='Q') exit(0);
  printf("%s",Message.Data);
  }
exit(0);
}
```

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.

1) Создание общей памяти.

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

int shmget (key_t key, int size, int shmemflg)
где
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.
shmemflg определяет флаги, управляющие поведением вызова.
В случае успешного завершения вызов возвращает положительное число – дескриптор области памяти, в случае неудачи - -1.

2) Доступ к разделяемой памяти.

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

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

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

где в **shmid** указан дескриптор.
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.

***shmaddr** виртуальный адрес в 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.

Shmflg - комбинация 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**.

2)
Открепление разделяемой памяти.

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

int shmdt(char *shmaddr)

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

3)
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)

где -
shmid -
дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти.

cmd – команда. 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.

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложения и снятия блокировки на нее и ее уничтожения.

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();
}
```

Массив семафоров

Семафоры 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).

1) Доступ к семафору

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

где **key** - ключ,

nsems - количество семафоров (длина массива семафоров)

semflag - 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.

2)Операции над семафором
int semop (int semid, struct sembuf *semop, size_t nops)
где

semid – идентификатор ресурса,

***semop** - указатель на структуру, определяющую операции, которые необходимо произвести над семафором.

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

nops - количество указателей на эту структуру, которые передаются функцией semop(). То есть операций может быть несколько и операционная система гарантирует их атомарное выполнение.
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).

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.

Таким образом, мы имеем критический участок в момент, когда один процесс еще не дописал строку, а другой ее уже читает. Поэтому следует установить некоторые синхронизации и задержки.

1й процесс:

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

int main(void)
{    key_t key;
     int semid, shmid;
     struct sembuf sops;
     char *shmaddr;
     char str[256];
     key = ftok(  /usr/mash/exmpl  ,  S  );      //      создаем
уникальный ключ
     semid  =  semget(key,1,0666  |  IPC_CREAT);  // создаем  один
семафор с определенными
     //        правами доступа
     shmid  =  shmget(key,256,  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,IPC_SET, (union semun) 0); //инициализируем
семафор со значением 0
     sops.sem_num = 0; sops.sem_flg = 0;

     // запуск бесконечного цикла
     while(1) { printf( Введите строку :  );
          if ((str = gets(str)) == NULL) break;
          sops.sem_op=0;                //   ожидание   обнуления
семафора
          semop(semid, &sops, 1);
          strcpy(shmaddr, str);             // копируем строку в
разд. память
          sops.sem_op=3;                // увеличение семафора на
3
          semop(semid, &sops, 1);
     }
```

```
     shmaddr[0]= Q  ;                    // укажем 2-ому процессу
на то,
     sops.sem_op=3;                                  // что пора
завершаться
     semop(semid, &sops, 1);
     sops.sem_op = 0;                    // ждем, пока обнулится
семафор
     semop(semid, &sops, 1);
     shmdt(shmaddr);                     // отключаемся от разд.
памяти
     semctl(semid, 0, IPC_RMID, (union semun) 0);     // убиваем
семафор
     shmctl(shmid, IPC_RMID, NULL);          //         уничтожаем
разделяемую память
     exit(0); }
```

2й процесс:

```
/* здесь нам надо корректно определить существование ресурса,
если он есть - подключиться, если нет - сделать что-то еще, но
как раз этого мы делать не будем */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(void)
{    key_t key; int semid;
     struct sembuf sops;
     char *shmaddr;
     char st=0;

     // далее аналогично предыдущему процессу   - инициализации
ресурсов
     semid = semget(key,1,0666 | IPC_CREAT);
     shmid = shmget(key,256, 0666 | IPC_CREAT);
     shmaddr = shmat(shmid, NULL, 0);
     sops.sem_num = 0; sops.sem_flg = 0;

     // запускаем цикл
     while(st!=  Q  ) {
          printf( Ждем открытия семафора  n  );
                              // ожидание положительного значения
семафора
          sops.sem_op=-2;
          semop(semid, &sops, 1);
          // будем ожидать, пока  значение семафора  +значение
sem_op  не перевалит за
                                   0, то есть если придет   3  то
  3-2=1
          // теперь значение семафора равно 1
          st = shmaddr[0];
```

```
        {    /*критическая секция - работа с разделяемой
    памятью - в этот момент
                                первый процесс к разделяемой памяти
    доступа не имеет*/}
                            /* после работы - закроем семафор */
         sem.sem_op=-1;
         semop(semid, &sops, 1);
                                // вернувшись в начало цикла мы
опять будем ждать,                              пока          значение
семафора не станет больше нуля
}
    shmdt(shmaddr); // освобождаем разделяемую память и выходим
    exit(0);
}
```

Механизм сокетов

Средства межпроцессного взаимодействия ОС Unix, представленные в системе IPC, решают проблему взаимодействия двух процессов, выполняющихся в рамках одной операционной системы.

Но необходимы:

• Унифицированный механизм, позволяющий использовать одни и те же подходы для локального и нелокального взаимодействия.

• Общий интерфейс, позволяющий пользоваться услугами различных протоколов по выбору пользователя.

Эти проблемы решает механизм сокетов (sockets)

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.

1)

Соединение с использованием виртуального канала (TCP) – 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.

2) Датаграммное соединение (UDP) 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:

- домен AF_UNIX для локального взаимодействия
- домен AF_INET для взаимодействия в рамках сети (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.

Далее мы рассмотрим принципиальный набор функций, которые предоставляются для работы с сокетами.

Для использования всех приведенных далее функций необходимо подключить следующие include-файлы:

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

1) Создание сокета

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

domain – константа, определяющая коммуникационный домен, который будет использоваться для взаимодействия, в частности значение AF_UNIX определяет домен локального межпроцессного взаимодействия внутри одной UNIX-системы, а значение AF_INET соответствует домену взаимодействия удаленных систем с использованием TCP/IP.

type определяет тип соединения (сокета). В частности type может принимать значения SOCK_STREAM, что соответствует сокету потока или виртуального канала, и SOCK_DGRAM, если мы хотим установить датаграммное соединение.

protocol – код протокола, который будет использоваться для взаимодействия. Значение IPPROTO_TCP используют, если тип сокета – виртуальный канал, IPPROTO_UDP – если речь идет о датаграммном сокете. Если protocol=0, то система сама подберет подходящий протокол.

Функция возвращает дескриптор сокета - некоторое положительное число в случае успешной работы и «-1» в случае возникновения проблем (в errno – причина).

2) Функция связывания

```
int bind(int sockfd, struct sockaddr * myaddr, int sizeaddr);
```

где **sockfd** – дескриптор сокета, который вернула функция socket.

sizeaddr – размер той структуры в байтах, на которую указывает второй аргумент (myaddr). Использование этого аргумента обусловлено тем, что реально в зависимости от домена форматы структур могут быть разными, например, в файле sys/un.h:

```
struct sockaddr_un
     {
     short sun_family;     // Код домена, в частности AF_UNIX
     char sun_path[108]; // имя файла, с которым будет связан адрес
     }
```

Суть работы: мы создали сокет и получили дескриптор сокета. Нам хотелось бы иметь возможность обращаться к этому сокету извне: из других процессов или даже других машин. Система предоставляет возможность связывания конкретного сокета с некоторым адресом.

Связывание необходимо для серверных программ и иногда используется в клиентских приложениях.

Функция bind возвращает «0» в случае успеха и «-1» в случае ошибки.

2) Функция предварительного соединения

int connect (int sockfd, struct sockaddr *servaddr, int addrlen);

где **sockfd** – дескриптор сокета.

servaddr – указатель на структуру, содержащую адрес сокета-сервера.

addrlen – длина структуры, содержащей адрес.

Суть дела состоит в том, что имеются разновидности сокетов с предварительной установкой соединения и без нее. Предварительная установка соединения требует наличия как клиента, так и сервера. Соответственно если тип сокета виртуальный канал, то предварительное соединение обязано иметь место. Если же тип соединения датаграмма, то можно обойтись и без предварительного соединения

Соединение осуществляется от имени клиента, т. е. в рамках сервера происходит регистрация того факта, что имеется клиент – сокет, который будет работать с сервером. Обращаю Ваше внимание, что установка соединения имеет смысл для домена INET.

3)Функции прослушивания сокета и подтверждения соединения

Рассмотрим серверные функции:

1) **int listen (int sockfd, int backlog);**

где **sockfd** – дескриптор сокета.

banklog – декларирование длины очереди запросов, которая будет ассоциирована с данным сокетом. Эта длина определяет количество необработанных запросов, понятно, что это число аккумулируется с ростом числа необработанных запросов, и может сложиться ситуация, когда свободного места для очередного необработанного запроса не будет. Так вот, реакция системы на факт исчерпания места зависит от типа используемого протокола.

Обращение к этой функции говорит о том, что процесс-сервер готов принимать запросы от клиентов, готов к работе с клиентами. До обращения к этой функции любой запрос к серверу не будет успешным.

2) функция подтверждения соединения.

int accept (int sockfd, struct sockaddr *clntaddr, int *addrlen);

где **sockfd** – дескриптор сокета.

clntaddr – указатель на структуру, в которой указан адрес клиента.

addrlen – указатель на целочисленную переменную, содержащую длину адреса (фактически, длину структуры sockaddr).

Эта функция создает дублирующий сокет, который будет связан с одним из клиентов. В случае успеха функция возвращает идентификатор сокета, это означает, что клиент после всех предварительных действий, которые он должен был выполнить, обращается с запросом на получение данных по адресу сокета-сервера. А в сервере работа с этим клиентом будет осуществлена только после того, как будет открыт дополнительный сокет (виртуальный канал с одним из клиентов, через который будет осуществляться передача информации). Соответственно, после того, как откроется виртуальный канал на взаимодействие с клиентом, могут осуществляться обмены следующих типов:

1. **read/write** – полностью аналогичны работе с файлами, но только вместо файлового дескриптора будет дескриптор сокета. Например, write в сервере

может работать с тем дескриптором сокета, который вернула функция accept, а в клиенте – с дескриптором, который вернула функция connect.

2. **send/recv** и **sendto/recvfrom** – реально эти функции Вы посмотрите сами в доступной литературе. В общем то, эти функции аналогичны read/write.

4) Функции закрытия сокета

1) **close – полная аналогия с работой с файлами, но аргумент – дескриптор сокета.**

2) int shutdown (int sd, int mode);

где **sd** – дескриптор сокета.

mode – некий модификатор, который определяет вариации закрытия сокета:

- на получение информации;
- на передачу информации;
- совсем.

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 */
```

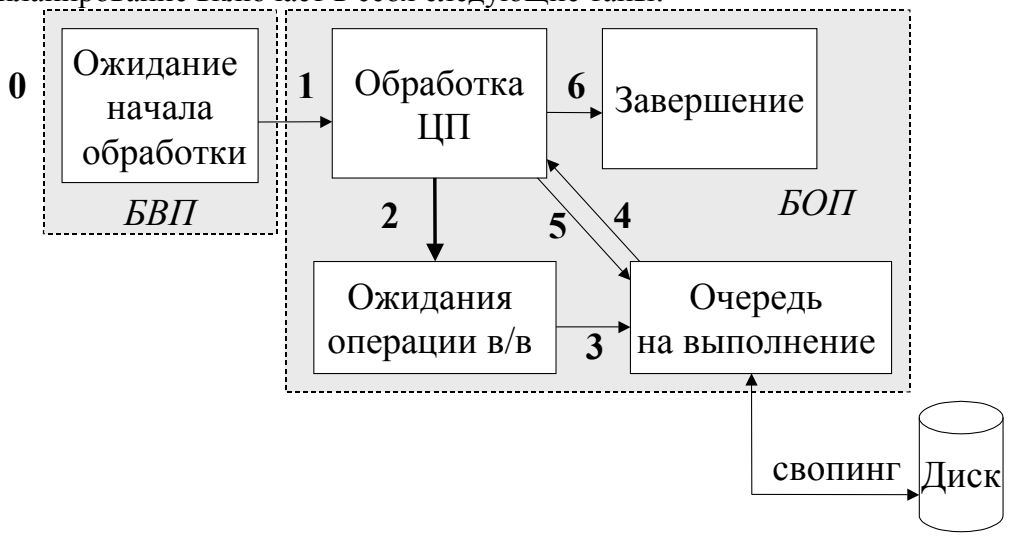
```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
     printf("can't create socket\n");

     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);
  }
     //  while (1)
}
```

Лекция 11. Планирование

Основные задачи планирования

планирование включает в себя следующие тапы:



- Планирование очереди процессов на начало обработки (проблема – принципы организации этой очереди)
- Планирование распределения времени ЦП между процессами (оптимальная загрузка процесса + обеспечение использования ресурсов каждым процессом, приоритеты. По идее, сюда же можно внести обработку прерываний)
- Планирование свопинга
- Планирование обработки прерываний
- Планирование очереди запросов на обмен
    - определение уровнЯ многопроцессности
    - Дисциплина обслуживания очереди :

1. простейшая – FIFO
2. по приоритету
3. с учетом предполагаемого времени выполнения процесса, объема операций ввода/вывода и так далее.

Основной проблемой планирования все же является планирование распределения времени ЦП между процессами . Раасмотрим основные особенности.

Квант времени – непрерывный период процессорного времени.
Приоритет процесса – числовое значение, показывающее степень привилегированности процесса при использовании ресурсов ВС (в частности, времени ЦП).

Для грамотного планирования надо решить две задачи:
– определить величину кванта
– определить стратегию обслуживания очереди готовых к выполнению процессов

Если величина кванта не ограничена – ***невытесняющая стратегия*** планирования времени ЦП (применяется в пакетных системах). Никто принудительно не скидывает процесс с ЦП. Разработчики берут на себя функции диспетчера.
Вытесняющая стратегия **-** величина кванта ограничена.

Рассмотрим, как решается проблема с определения кванта времени.

Кванты постоянной длины.

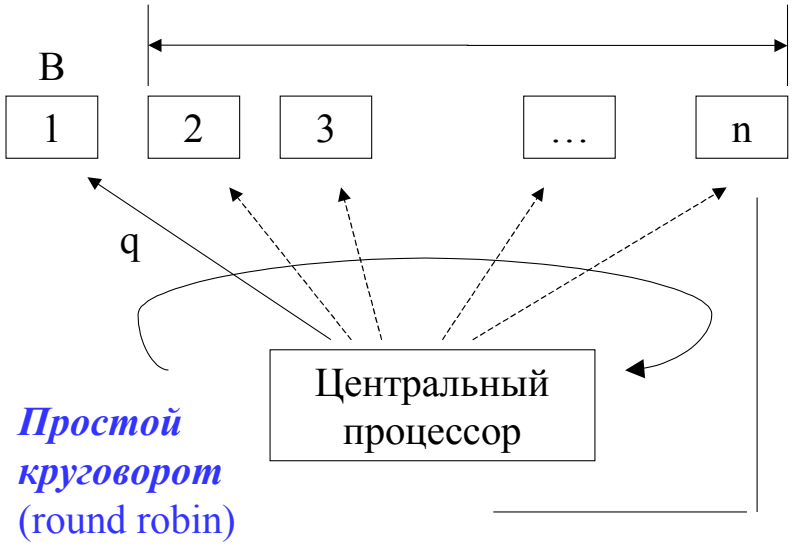
•Время ожидания кванта процессом ~ q(n-1)

•Параметры: длина очереди и величина кванта.

•Дисциплина обслуживания очереди, например, FIFO.

•Переключение процессов – операция, требующая времени.

Проблема: как определить длину кванта. Слишком маленький – не хватит времени на переключение, большой - некоторые успеют выполниться полностью.



Простой круговорот (round robin)

Кванты переменной длины

Величина кванта может меняться со временем

• Вначале «большой» квант q=A,на следующем шаге q=A-t, q=A-2t,…, до q=B (B<A). Преимущество для коротких задач.

• Вначале q=B, далее q=B+t,…, до q=A. Уменьшение накладных расходов на переключение задач, когда несколько задач выполняют длительные вычисления.

Если процесс интенсивно пользуется операциями ввода/вывода, то он может использовать выделенный квант не до конца.

Алгоритмы, основанные на приоритетах

Вычисление приоритета основывается на статических и динамических характеристиках. Изменение приоритета может происходить по инициативе процесса, пользователя, ОС. Правила назначения приоритета процессов определяют эффективность работы системы.

Планирование по наивысшему приоритету (highest priority first - HPF).

При появлении в очереди готовых процессов процесса с более высоким приоритетом, чем у текущего наступает момент смены процесса.
Возможно два варианта:

- относительный приоритет (ожидание исчерпания кванта у текущего процесса)
- абсолютный приоритет (немедленная смена текущего процесса)

Пример использования стратегии HPF.
Выбор самого короткого задания (shortest job first - SJF).

Время выполнения – характеристика, на которой основан приоритет. Приоритет обратно пропорционален ожидаемому времени обработки.
Этот вариант
удобен для “коротких” процессов.

1) Класс подходов, использующих линейно возрастающий приоритет.

Процесс при входе в систему получает некий приоритет, который возрастает с коэффициентом A во время ожидания в очереди готовых процессов, и с коэффициентом B во время выполнения.
Из выбора A и B - разные правила планирования:

- Если 0<A<=B обслуживание очереди по дисциплине FIFO
- Если 0>B>=A обслуживание очереди по дисциплине LIFO

Нелинейные функции изменения приоритета

Например, приоритет убывает по линейному закону с течением времени. Когда достигается некое максимальное время, приоритет скачком возрастает до некоторой большой величины. Это благоприятствует коротким процессам, и при этом соблюдается условие, что ни одному процессу не придется ждать обслуживания слишком долго.

Разновидности круговорота.

Простой круговорот (RR – round robin) не использует никакой статистической или динамической информации о приоритетах. (см. рисунок выше)

При ***круговороте со смещением*** каждому процессу соответствует своя длина кванта, пропорциональная его приоритету.

«Эгоистический» круговорот. Если параметры A и B : 0<=B<A.
Процесс, войдя в систему ждет пока его приоритет не достигнет приоритета работающих процессов, а далее выполняется в круговороте.

Приоритет выполняемых процессов увеличивается с коэффициентом B<A, следовательно, ожидающие процессы их догонят.
При B=0 «эгоистический» круговорот практически сводится к простому.

Очереди с обратной связью (feedback – FB).

Используется N очередей. Новый процесс ставится в первую очередь, после получения кванта он переносится во вторую и так далее. Процессор обслуживает непустую очередь с наименьшим номером.
В FB поступивший процесс **неявно** получает наивысший приоритет и выполняется подряд в течении нескольких квантов до прихода следующего, но не более чем успел проработать предыдущий.
«-» Работа с несколькими очередями – издержки.
«+» Удобны для коротких заданий: не требуется предварительная информация о времени выполнения процессов.

Смешанные алгоритмы планирования

На практике концепции квантования и приоритетов часто используются совместно.
К примеру, в основе – концепция квантования, а определение кванта и/или дисциплина обслуживания очередей базируется на приоритетах.

Планирование в системах реального времени

Системы реального времени являются специализированными системами в которых все функции планирования ориентированы на обработку некоторых событий за время, не превосходящее некоторого предельного значение.

Системы реального времени бывают “Жесткие” и ”мягкие ”.

В первом случае время завершения выполнения каждого из процессов должно быть ***гарантировано*** для всех сценариев функционирования системы.
Это может быть обеспечено за счет :
- полного тестирования всевозможных сценариев

- построения статического расписания

- выбора математически просчитанного алгоритма динамического планирования

Периодические запросы – все моменты запроса периодического процесса можно определить заранее.
Пусть $\{T_i\}$ набор периодических процессов с периодами – p_i , предельными сроками выполнения d_i и требованиями ко времени выполнения c_i.
Для проверки возможного составления расписания анализируется расписание на отрезке времени равному наименьшему общему множителю периодов этих процессов.
Необходимое условие наличия расписания:
Сумма коэффициентов использования $\mu=\Sigma\ c_i / p_i <= k$, где k - количество доступных процессоров.

2)Алгоритмы с динамическим изменением приоритетов.

Параметр ***deadline*** – конечный срок выполнения.
Выбор процесса на выполнение по правилу:
выбирается процесс, у которого текущее значение разницы между конечным сроком выполнения и временем, необходимым для его непрерывного выполнения, является наименьшим.

Таким образом, можно сфтрмулировать общие критерии для сравнения алгоритмов планирования

- использование времени ЦП
- пропускная способность (кол-во процессов в единицу времени)
- время ожидания (в очереди готовых)
- время оборота (полное время от момента поступления до завершения)
- время отклика (для интерактивных программ – время от поступления в систему до момента первого обращения к терминалу
- предельный срок выполнения процесса
- и т.д.
-

Планирование в ОС UNIX

Используется принцип кругового планирования в рамках очередей каждого приоритета.
Если процесс не завершается или не блокируется в рамках 1 секунды – он вытесняется.
В общем случае значение приоритета есть функция
P=F (CPU, nice), т.е. в вычислении приоритета используются две изменяемые составляющие – CPU (системная) и nice (пользовательская). Учитывается история выполнения, величины CPU и nice ограничены.

Пересчет приоритета процесса происходит в момент выбора процесса для выполнения на ЦП 1 раз в секунду.
Процессам назначается базовый приоритет, чтобы их можно было разделять на фиксированные группы уровней приоритетов.
Эти группы используются для оптимизации доступа к блочным устройствам (например, к диску) и обеспечения быстрого отклика операционной системы на системные вызовы.
Группы приоритетов
(в порядке убывания)
- программа свопинга
- управление байт-ориентированными устройствами ввода/вывода
- пользовательские процессы

Иерархия обеспечивает эффективное использование устройств ввода/вывода

Используются формулы: $CPU_j\ (i) = \frac{CPU_j\ (i-1)}{2}$

$$P_j\ (i) = Base_j + \frac{CPU_j\ (i-1)}{2} + nice_j$$

CPU_j (i) - время использования ЦП процессом j за время i;

P_j (i) - приоритет процесса j в начале кванта i (приоритет выше, если значение меньше);

$Base_j$ - базовый приоритет j-го процесса (необходим для разделения процессов на фиксированные группы уровней приоритетов);

$nice_j$ - пользовательская составляющая приоритета (значение может только увеличиваться до некоторого уровня).

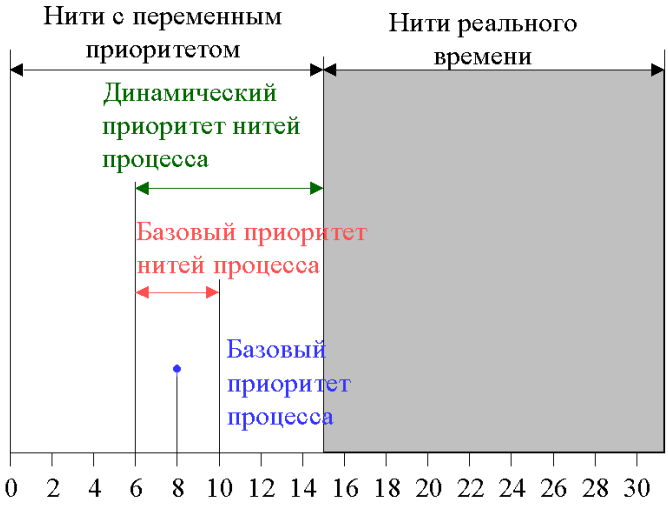
Пример традиционного планирования процессов в ОС Unix
В примере не учитывается составляющая nice.

	Процесс A		Процесс B		Процесс C	
Время	Приоритет	Счетчик	Приоритет	Счетчик	Приоритет	Счетчик
0	60	0 → 60	60	0	60	0
1	75	30	60	0 → 60	60	0
2	67	15	75	30	60	0 →60
3	63	7 → 67	67	15	75	30
4	76	33	63	7 → 67	67	15
5	68	16	76	33	63	7

Планирование в Windows NT.

Квантование сочетается с использованием динамических абсолютных приоритетов.
В системе определено 32 уровня приоритетов.
Два класса нитей:
Нити с переменными приоритетами (0-15]
Нити "реального" времени (16-31] – высокоприоритетные нити.
Критичны по времени выполнения.

Нити с переменными приоритетами
Изначально процессу присваивается базовый приоритет.
Базовый приоритет процесса может меняться ОС, следовательно, могут измениться базовые приоритеты составляющих его нитей.
Нить получает значение приоритета из диапазона базового приоритета.
Приоритет нити может отклоняться от своего базового приоритета, и это может быть не связано с изменением базового приоритета процесса (см. диапазон значений динамического приоритета нитей).



• Поддерживается группа очередей (для нитей с переменными приоритетами) – по одной для каждого приоритета. Система просматривает очереди, начиная с самой приоритетной.
• На выполнение выбирается нить с наивысшим приоритетом. Ей выделяется квант времени. Если во время выполнения в очереди появляется нить с более высоким приоритетом, то текущая нить вытесняется. Вытесненная нить становится в очередь готовых впереди тех, что имеют тот же приоритет.
• Если нить исчерпала квант – ее приоритет понижается на единицу и она перемещается в соответствующую очередь
• Повышается значение приоритета – при выходе из состояния ожидания окончания ввода-вывода

Планирование свопинга в ОС Unix

При принятии нового процесса на обработку необходимо освободить ресурсы. Какой - то процесс скидывается в область своппинга.
Область свопинга - специально выделенное системой пространство внешней памяти

P_TIME – счетчик, находящийся в контексте процесса. Суммирует время нахождения процесса в состоянии мультипрограммной обработки или в области свопинга. При переходе из одного состояния в другое счетчик обнуляется. 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**.

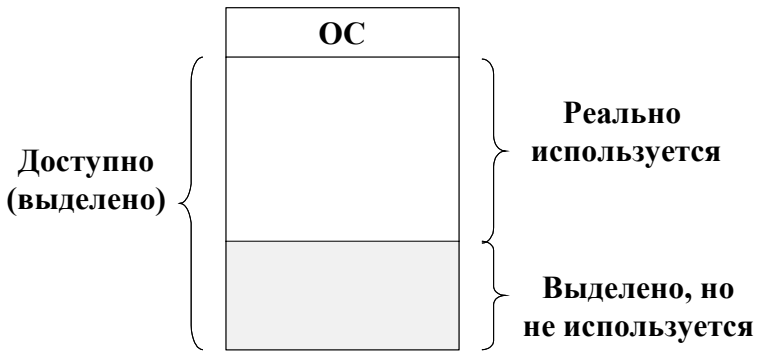
Лекция 12. Управление оперативной памятью

Основные задачи:
1.Контроль состояния каждой единицы памяти (система должна отслеживать, какая единица свободна/распределена, обеспечение аппаратурой кр==омпьютера и ОС - таблицы).
2.Стратегия распределения памяти (выбор правил, по которым принимаются решения, кому, когда и сколько памяти должно быть выделено).
3.Выделение памяти (принятие решения: выбор конкретной области, которая должна быть выделена).
4.Стратегия освобождения памяти (процесс освобождает, ОС "забирает" окончательно или временно. Здесь же выбор стратегии).

Рассмотрим стратегии управления по следующему плану

1.Основные концепции.
2.Необходимые аппаратные средства.
3.Основные алгоритмы.
4.Достоинства, недостатки.

1.Одиночное непрерывное распределение.

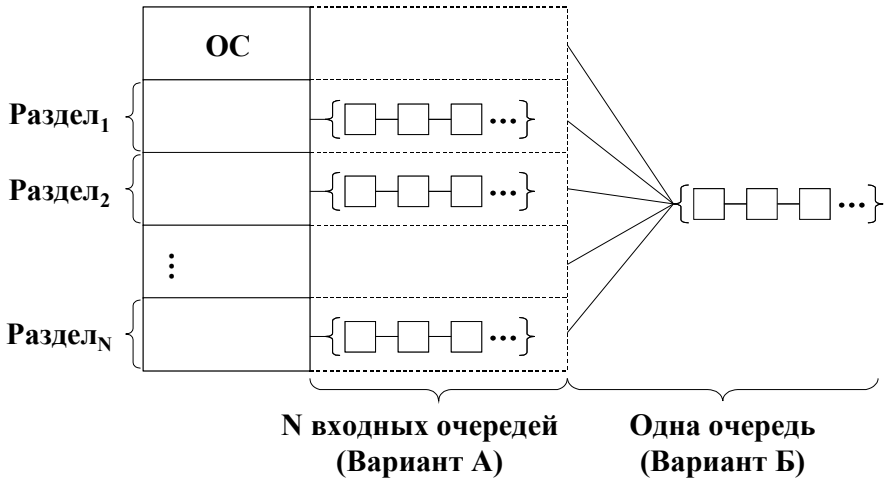


Необходимые аппаратные средства:
•Регистр границ + режим ОС / режим пользователя.
•Если ЦП в режиме пользователя попытается обратиться в область ОС, то возникает прерывание.

Алгоритмы: очевидны.

Достоинства: простота.

Недостатки:
1.Часть памяти не используется.
2.Процессом/заданием память занимается все время выполнения.
3.Ограничение на размеры задания.



2.Распределение разделами.

Необходимые аппаратные средства:

1.Два регистра границ.
Недостатки:
а. перегрузка регистра границ при каждой смене контекста;
б. сложности при использовании каналов/процессоров ввода/вывода.
2.Ключи защиты (PSW).

Алгоритмы: *Модель статического определения разделов*

А. Сортировка входной очереди процессов по отдельным очередям к разделам. Процесс размещается в разделе минимального размера, достаточного для размещения данного процесса. В случае отсутствия процессов в каких-то под очередях – неэффективность использования памяти.

Алгоритмы: *Модель статического определения разделов*
Б. Одна входная очередь процессов.
1. Освобождение раздела поиск (в начале очереди) первого процесса, который может разместиться в разделе. Проблема: большие разделы маленькие процессы.

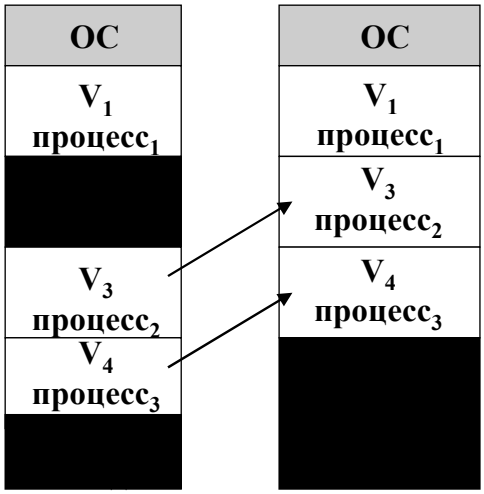
2. Освобождение раздела поиск процесса максимального размера, не превосходящего размер раздела.
Проблема: дискриминация “маленьких” процессов.

4. Оптимизация варианта 2. Каждый процесс имеет счетчик дискриминации. Если значение счетчика процесса ³ K, то обход его в очереди невозможен.

Достоинства:
1.Простое средство организации мультипрограммирования.
2.Простые средства аппаратной поддержки.
3.Простые алгоритмы.

Недостатки:
1.Фрагментация.
2.Ограничение размерами физической памяти.
3.Весь процесс размещается в памяти – возможно неэффективное использование.

3.Распределение перемещаемыми разделами.



Виртуальная память

Процесс$_4$

(например, V= V_2+1/2V_5)

Необходимые аппаратные средства:
1.Регистры границ + регистр базы
2.Ключи + регистр базы
Алгоритмы:
Аналогично предыдущему
Достоинства:
1.Ликвидация фрагментации
Недостатки:
1.Ограничение размером физической памяти
2.Затраты на перекомпоновку

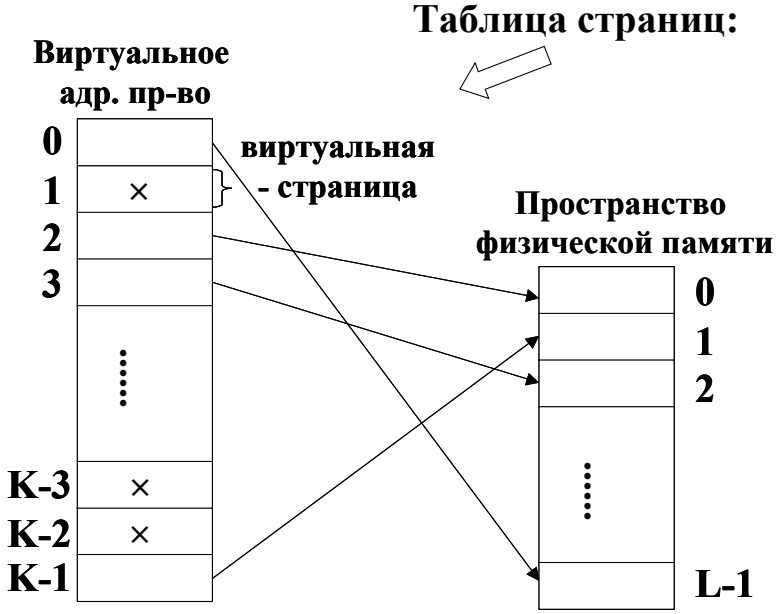
4.Страничное распределение.

Основные концепции:

Таблица страниц – отображение номеров виртуальных страниц на номера физических.

Проблемы:

1. Размер таблицы страниц (количество 4кб страниц при 32-х разрядной адресации – 1000000. Любой процесс имеет собственную таблицу страниц).
2. Скорость отображения.



Необходимые аппаратные средства:

1.Полностью аппаратная таблица страниц (стоимость, полная перегрузка при смене контекстов, скорость преобразования).

2.Регистр начала таблицы страниц в памяти (простота, управление смены контекстов, медленное преобразование).
3.Гибридные решения.

Алгоритмы и организация данных:

Размеры таблицы страниц – иерархическая организация таблицы страниц.

Модельная структура записи таблицы страниц

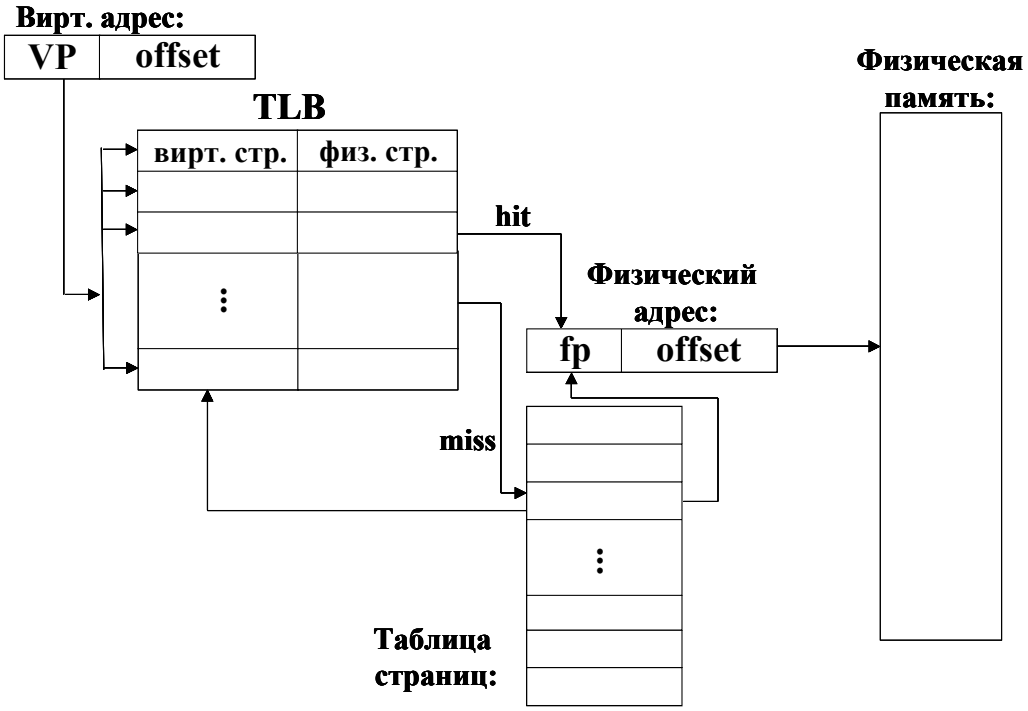
α – присутствие/отсутствие
β – защита (чтение, чтение/запись, выполнение)
γ – изменения
δ – обращение (чтение, запись, выполнение)
ε – блокировка кэширование

TBL (Translation Lookaside Buffer) – буфер быстрого преобразованич адресов.

Суть: для разрешенич коммуникаций, связанных со скоростью и т.д. необходимо гибридное решение. Имеется аппаратный буфер (т.н. буфер ассоциативной памяти) относительно небольшого размера, кот используется в качестве КЭШа таблицы страниц.
Структура этого буфера:

- N виртуальных страниц
- N соответствующих физических страниц

Если попадаем – автоматичекая замена поля физической страницы виртуальной, иначе – прерывание, по которому управление передается ОС и она уже сама обновляется.



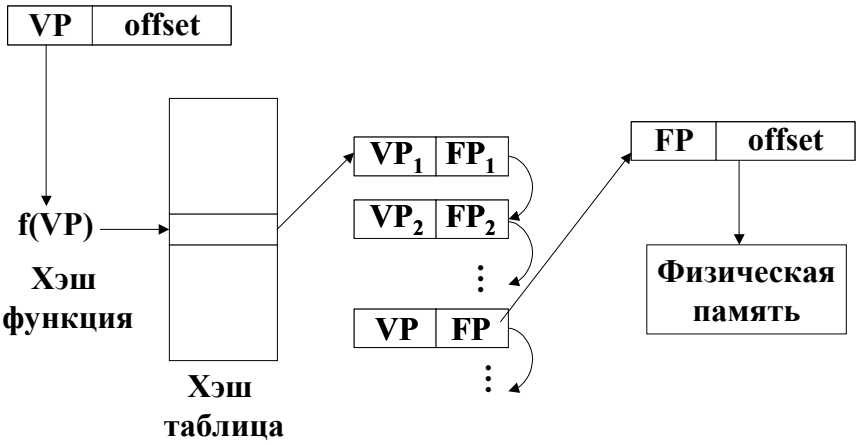
Проблема: стратегии по преобразованию строчки в TLB => организпция алгоритмов выбора.

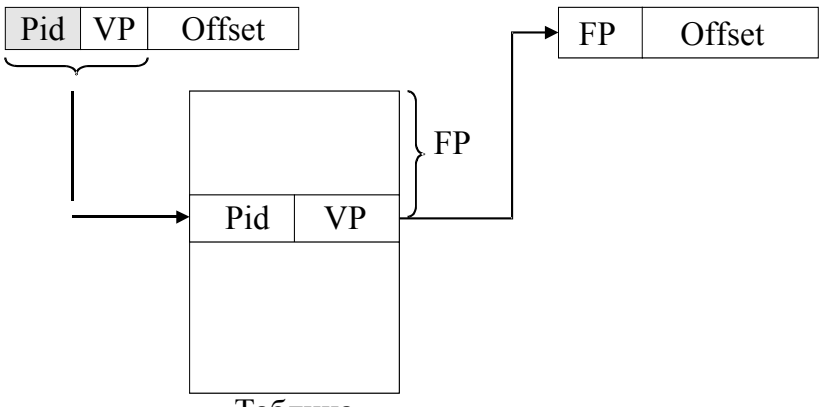
Длугая проблема – объем таблицы страниц.
Иерархия организации таблицы страниц.

1) Многоуровневая:
- индексация по внешней таблице страниц
- смещение по этой странице

2) hash-таблицы, которые

•Обычно используются для адресации
•Используется больше 32 разрядов.





3) инвертированные таблицы страниц

Проблема – поиск по таблице
Использование хэширования

Замешение страниц
Проблема загрузки «новой» страницы в память. Необходимо выбрать страницу для удаления из памяти (с учетом ее модификации и пр.)

Существуют разные алгоритмы:

1. **Алгоритм NRU**

(Not Recently Used – не использовавшийся в последнее время), который_спользует

R - обращение
M - изменение
} устанавливаются аппаратно

обнуление – программно (ОС)

биты статуса страницы в записях таблицы страниц

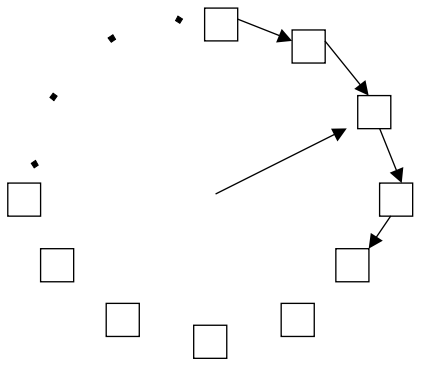
1.При запуске процесса M и R для всех страниц процесса обнуляются
2.По таймеру происходит обнуление всех битов R
3.При возникновении страничного прерывания ОС делит все страниц на классы:
•Класс 0:
•Класс 1:
•Класс 2:
•Класс 3:
4.Случайная выборка страницы для удаления в непустом классе с минимальным номером

Стратегия: лучше выгрузить измененную страницу, к которой не было обращений как минимум в течение 1 «тика» таймера, чем часто используемую страницу

2. **Алгоритм FIFO**

«Первым прибыл – первым удален» - простейший вариант FIFO. (проблемы «справедливости»)

Модификация алгоритма (алгоритм вторая попытка):
1.Выбирается самая «старая страница». Если R=0, то она заменяется
2.Если R=1, то R – обнуляется, обновляется время загрузки страницы в память (т.е. переносится в конец очереди). На п.1



3. **Алгоритм «Часы»**

1.Если R=0, то выгрузка страницы и стрелка на позицию вправо.
2.Если R=1, то R-обнуляется, стрелка на позицию вправо и на П.1.

4. **Алгоритм LDU**

(Least Recently Used – «менее недавно» - наиболее давно используемая страница)

Вариант возможной аппаратной реализации.
Памяти N – страниц.
Битовая матрица NxN (изначально все биты обнулены).
При каждом обращении к i<u>ой</u> странице происходит присваивание 1 всем битам i<u>ой</u> строки и обнуление все битов i<u>го</u> столбца.

Строка с наименьшим 2^{ным} кодом соответствует искомой странице.

(аппаратная реализация крайне тяжелая).

5. **Алгоритм NFU**

(Not Frequently Used – редко использовавшаяся страница)
Программная модификация LRU.
Для каждой физической страницы i – программный счетчик $Count_i$
0. Изначально $Count_i$ – обнуляется для всех i.
1.По таймеру $Count_i = Count_i + R_i$
Выбор страницы с минимальным значением $\{Count_i\}$
<u>Недостаток</u> – «помнит» всю активность по использованию страниц

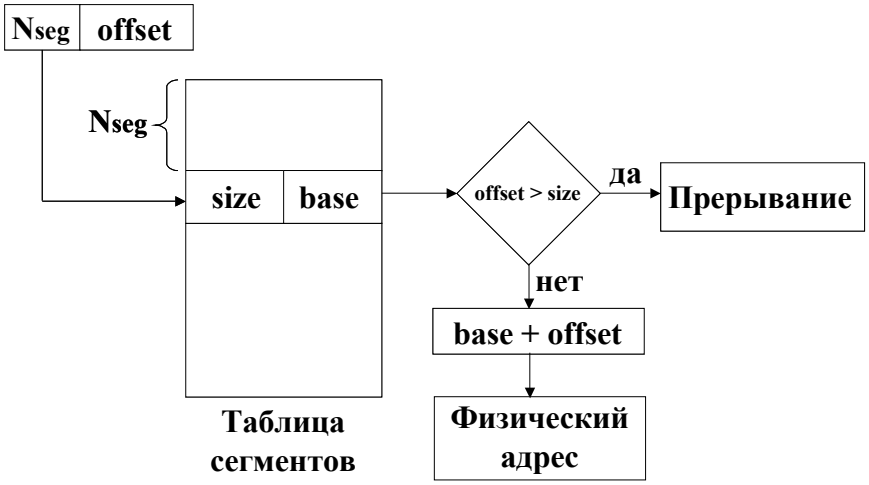
6. **Модификация NFU**

– алгоритм старения
<u>Модификация:</u>
1.Значение счетчика сдвигается на 1 разряд вправо.
2.Значение R добавляется в крайний левый разряд счетчика.

4. Сегментная организация памяти.

<u>Основные концепции:</u>
•Виртуальное адресное пространство представляется в виде совокупности сегментов
•Каждый сегмент имеет свою виртуальную адресацию (от 0 до N-1)
•Виртуальный адрес: <номер_сегмента, смещение>

Необходимые аппаратные средства:



5. Сегментно-страничная организация памяти.

Необходимые аппаратные средства:

Упрощенная модель Intel.

Необходимые аппаратные средства:

Достоинства\недостатки как у страничной, + достоинства сегментной и сегментно-страничной.

Оглавление.