

## Сроки

- Подача документов **2020-06-20 по 2020-08-06**
- Экзамены **2020-07-30 и 2020-08-14 августа** 

## Ссылки

- [Поступление](#)
- [Билеты ФИТ 9.04.01 «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»](#)

# ОГЛАВЛЕНИЕ

1 .....	13
1.1 Понятие архитектуры вычислительных систем (ВС). .....	13
1.2 Классификация ВС.....	14
1.2.1 По назначению .....	14
1.2.2 По типу.....	15
1.2.3 По типу ЭВМ или процессоров.....	17
1.2.4 По степени территориальной разобщенности .....	17
1.3 Принципы организации CISC и RISC архитектур.....	19
1.3.1 Вместе .....	19
1.3.2 RISC.....	20
1.3.3 CISC.....	21
1.3.4 MISC.....	22
2 .....	22
2.1 Многопроцессорные системы. Симметричная и асимметричная многопроцессорность. .....	22
2.1.1 Симметрическая многопроцессорность.....	22
2.1.1.1 Википедия.....	28
2.1.1.2 Описание.....	28
2.1.1.3 Преимущества и недостатки.....	30
2.1.1.4 Ограничение на количество процессоров .....	31
2.1.1.5 Проблема когерентности кэш-памяти .....	31
2.1.1.6 Поддержка операционной системой.....	31
2.1.1.7 Альтернативы.....	32
2.1.2 Асимметрическая многопроцессорность.....	32
2.1.2.1 Википедия.....	33

3 .....	35
3.1 Методы организации сетей ЭВМ .....	35
3.2 Основные принципы их функционирования. ....	36
3.3 Классификация сетей по масштабу и топологии.....	36
3.3.1 Искусственные и реальные сети.....	36
3.3.2 Территориальная распространенность .....	37
3.3.3 Тип среды передачи информации .....	38
3.3.4 Топология компьютерных сетей .....	38
3.3.4.1 Линейная сеть.....	38
3.3.4.2 Кольцевая сеть .....	39
3.3.4.3 Звездообразная сеть.....	39
3.3.4.4 Общая шина.....	39
3.3.4.5 Древовидная сеть .....	39
3.3.4.6 Ячеистая сеть.....	40
3.3.4.7 Полносвязная сеть .....	40
3.3.5 Одноранговые и иерархические сети.....	40
3.3.5.1 Одноранговые сети.....	40
3.3.5.2 Иерархические сети.....	40
3.3.5.3 Две технологии использования сервера .....	41
3.4 Понятие сетевого протокола.....	42
3.4.1 Пример .....	43
3.5 Семиуровневая модель OSI/ISO.....	44
3.5.1 Уровень 1. Физический. ....	45
3.5.2 Уровень 2. Канальный.....	45
3.5.3 Уровень 3. Сетевой. ....	46
3.5.4 Уровень 4. Транспортный. ....	46
3.5.5 Уровень 5. Сеансовый. ....	47
3.5.6 Уровень 6. Представления данных. ....	48
3.5.7 Уровень 7. Прикладной.....	48
3.6 Способы маршрутизации сообщений в сетях ЭВМ .....	49
3.7 Сетевая архитектура TCP/IP: основные принципы организации и функционирования.....	49
3.7.1 Уровни стека TCP/IP .....	50

3.7.2 Прикладной уровень.....	51
3.7.3 Транспортный уровень .....	51
3.7.4 Сетевой (межсетевой) уровень.....	52
3.7.5 Канальный уровень.....	53
4 .....	53
4.1 Система прерываний.....	54
4.1.1 Время реакции.....	55
4.1.2 Глубина прерывания.....	56
4.1.3 Типы прерываний .....	56
4.1.3.1 Аппаратные .....	56
4.1.3.2 Программные .....	56
4.1.4 Обработчик прерывания .....	58
4.2 Защита памяти .....	58
4.2.1 Защита отдельных ячеек .....	59
4.2.2 Метод граничных регистров.....	60
4.2.3 Метод ключей защиты .....	60
4.2.4 Защита по привилегиям.....	61
4.3 Механизм преобразования адресов в системах виртуальной памяти .....	62
4.3.1 Статическое распределение .....	62
4.3.2 Динамическое распределение.....	63
4.3.3 Страницы .....	64
4.4 Управление периферийными устройствами .....	65
4.4.1 Программно-управляемая передача.....	67
4.4.2 Прямой доступ к памяти (DMA) .....	68
4.5 Операционная система .....	70
4.5.1 Процесс .....	72
4.5.2 Очередь .....	73
4.5.3 ОС – процесс .....	73
5 .....	75
5.1 Стратегии управления оперативной памятью и Виртуальная память .....	75
5.1.1 Требования к стратегиям .....	75
5.1.1.1 Распределение .....	75

5.1.1.2 Защита.....	75
5.1.1.3 Разделение .....	75
5.1.1.4 Логическая организация .....	76
5.1.1.5 Физическая организация.....	76
5.1.2 Организация памяти .....	76
5.1.2.1 Единое постоянного использования .....	76
5.1.3 Разделение памяти .....	77
5.1.3.1 Однаковые участки.....	77
5.1.3.2 Неравные разделы.....	77
5.1.3.3 Динамическое разделение.....	78
5.1.3.4 Подкачка страниц .....	78
5.1.3.5 Виртуальная память.....	80
5.1.3.6 Подгрузка фреймов.....	80
5.1.3.7 Copy on write .....	80
5.1.4 Пробуксовка .....	81
5.2 Статическая и динамическая сборка.....	81
5.2.1 Статическая библиотека.....	81
5.2.2 Динамическая библиотека .....	82
5.2.3 Библиотека импорта .....	83
5.2.4 Короткое объяснение.....	83
6 .....	84
6.1 Распределение и использование ресурсов вычислительной системы и управление ими. ....	84
6.1.1 Долгосрочное планирование .....	86
6.1.2 Среднесрочное планирование .....	86
6.1.3 Краткосрочное планирование.....	86
6.2 Основные подходы и алгоритмы планирования.....	87
6.2.1 Приоритеты .....	87
6.2.2 Функция выбора.....	88
6.2.2.1 Первый зашел – первый стал обслужен .....	89
6.2.2.2 Циклический .....	89
6.2.2.3 Следующий - кратчайший .....	89
6.2.2.4 Следующий – тот, кому осталось меньше времени .....	90

6.3 Системы реального и разделенного времени.....	90
6.3.1 Системы реального времени.....	90
6.3.2 Архитектуры.....	90
6.3.2.1 Монолитная архитектура. ....	90
6.3.2.2 Уровневая (слоевая) архитектура. ....	91
6.3.2.3 Архитектура «клиент-сервер».....	91
6.3.3 Особенности ядра .....	92
6.3.4 Отличия от операционных систем общего назначения .....	94
6.3.5 Работа планировщика.....	94
6.3.6 Выполнение задачи.....	95
6.3.7 Алгоритмы планирования.....	96
6.3.8 Взаимодействие между задачами и разделение ресурсов .....	96
6.3.9 Выделение памяти .....	97
7 .....	98
7.1 Взаимодействие процессов. ....	98
7.1.1 Файл.....	99
7.1.2 Сигнал .....	99
7.1.3 POSIX .....	99
7.1.4 UNIX.....	100
7.1.5 Сокет.....	100
7.1.6 Именованный канал.....	101
7.1.7 Канал .....	101
7.1.8 Семафор .....	102
7.1.9 Разделяемая память .....	103
7.1.10 Проецируемый в память файл mmap .....	105
7.1.11 Очередь сообщений .....	106
7.1.12 Мэйлслот.....	106
7.1.13 Видеолекция .....	107
7.2 Разделяемая память, средства синхронизации. ....	108
7.3 Очереди сообщений и другие средства обмена данными. ....	108
8 .....	108
8.1 Управление доступом к данным.....	108

8.1.1 Избирательное (дискреционное) разграничение .....	108
8.1.1.1 Википедия (простое объяснение).....	108
8.1.1.2 Другое объяснение .....	110
8.1.2 Мандатное разграничение.....	111
8.1.3 Ролевое разграничение .....	114
8.1.4 Презентация.....	115
8.2 Файловые системы.....	117
8.2.1 Типы файлов.....	119
8.2.2 Атрибуты файлов.....	121
8.2.3 Особенности файловых систем .....	121
8.2.4 Реализация ФС .....	122
8.2.4.1 Выделение непрерывной последовательностью блоков .....	122
8.2.4.2 Выделение связным списком .....	123
8.2.4.3 Распределение связным списком с использованием индекса .....	124
8.2.4.4 Индексные узлы .....	125
8.2.5 RAID.....	127
8.2.5.1 Raid 0.....	127
8.2.5.2 Raid 1 .....	128
8.2.5.3 Raid 2 .....	128
8.2.5.4 Raid 3 .....	129
8.2.5.5 Raid 4 .....	130
8.2.5.6 Raid 5 .....	130
8.2.5.7 Raid 6.....	132
8.2.5.8 Raid 10.....	133
8.2.6 Вики.....	133
9 .....	135
9.1 Языки программирования .....	135
9.2 Концепции процедурно-ориентированного программирования .....	136
9.3 Объектно-ориентированного программирование .....	137
9.4 Функциональное программирование.....	138
9.5 Логическое программирование .....	139
9.6 Раннее (статическое) и позднее (динамическое) связывание.....	142

9.7 Статическая и динамическая типизация.....	142
9.7.1 Статически / динамически типизированные языки .....	142
9.7.2 Сильная / слабая типизация .....	143
9.7.3 Явная / неявная типизация .....	144
9.7.4 Итог .....	144
10 .....	145
10.1 Понятие о методах трансляции .....	145
10.2 Лексический анализ .....	145
10.3 Синтаксический анализ .....	146
10.3.1 Восстановление в режиме паники.....	146
10.3.2 Восстановление на уровне фразы .....	147
10.3.3 Нисходящий синтаксический анализ.....	147
10.4 Семантический анализ.....	148
10.5 Основные алгоритмы генерации объектного кода .....	149
10.5.1 Многоадресный код с неявно именуемым результатом (триады)....	150
10.5.2 Свертка объектного кода.....	151
10.5.3 Исключение лишних операций .....	152
10.5.3.1 Триады .....	152
10.5.3.2 SSA .....	156
10.5.4 Синтаксически управляемый (СУ) перевод.....	159
10.5.5 Шлак .....	160
10.6 Машинно-ориентированные языки (ассемблеры).....	160
10.7 Макросредства, макровызовы, языки макроопределений, условная макрогенерации принципы реализации .....	161
10.7.1 Макросредства .....	161
10.7.2 Макровызовы.....	162
10.7.3 Условная макрогенерация .....	163
10.8 Системы программирования, типовые компоненты .....	163
10.8.1 Редактор текста .....	163
10.8.2 Трансляторы .....	163
10.8.3 Компоновщик, или редактор связей .....	164
10.8.4 Отладчик .....	164
10.8.5 Загрузчик .....	164

11 .....	165
11.1 Принципы модульного, компонентного, объектно-ориентированного проектирования .....	165
11.1.1 Принципы модульного проектирования .....	165
11.1.2 ?Принципы компонентного проектирования .....	165
11.1.3 Принципы объектно-ориентированного проектирования.....	166
11.2 Шаблоны проектирования. ....	166
11.2.1 Шаблон «Стратегия» .....	167
11.2.2 Шаблон «Адаптер» .....	167
11.2.3 Шаблон «объектный пул».....	168
11.2.4 Шаблон «Итератор».....	168
11.2.5 Шаблон «Фабричный метод» .....	168
11.2.6 Шаблон «Декоратор».....	168
11.2.7 Шаблон «Одиночка».....	168
11.2.8 Шаблон «Делегирование».....	169
11.3 Моделирование программных систем, язык UML. ....	169
11.3.1 Class .....	171
11.3.2 Object .....	171
11.3.3 Package .....	172
11.3.4 Model .....	172
11.3.5 UseCase.....	173
11.3.6 Activity.....	175
11.3.7 Sequence .....	176
11.3.8 Deployment.....	177
11.4 Современные подходы к автоматическому синтезу программ.....	178
12 .....	179
12.1 Современные методы и технологии построения распределённых программных систем .....	179
12.1.1 J2EE .....	181
12.1.1.1 Достоинства.....	183
12.1.1.2 Недостатки.....	183
12.1.2 .NET Framework .....	183
12.1.2.1 Windows Forms.....	184

12.1.2.2 WPF .....	185
12.1.2.3 ADO.NET .....	186
12.1.2.4 ASP.NET .....	187
12.1.3 Web-сервисы.....	187
12.1.4 CORBA.....	188
13 .....	188
13.1 Концепция типа и моделей данных.....	188
13.1.1 ER-модель / сущность-связь .....	191
13.1.1.1 один ко многим (1 : n) .....	194
13.1.1.2 многое к одному (n : 1).....	195
13.1.1.3 многие ко многим (n : n). .....	195
13.1.1.4 Один к одному (1 : 1).....	196
13.1.2 Иерархическая модель данных.....	197
13.1.3 Сетевая модель данных.....	200
13.1.4 Реляционная модель данных. ....	202
13.1.5 Объектно-ориентированные БД .....	202
13.2 Абстрактные типы данных.....	206
13.3 Объекты (основные свойства и отличительные черты).....	207
14 .....	207
14.1 Основные структуры данных.....	207
14.1.1 Массивы.....	207
14.1.2 Стеки .....	208
14.1.3 Очереди.....	209
14.1.4 Связанный список.....	210
14.1.5 Ассоциативный массив .....	211
14.1.6 Графы .....	211
14.1.7 Деревья.....	213
14.1.7.1 Бинарное дерево.....	215
14.1.7.2 Дерево Бинарного Поиска .....	216
14.1.7.3 Сбалансированное дерево.....	217
14.1.7.4 N дерево .....	219
14.1.7.5 AVL дерево ( <del>сбалансированное дерево</del> ) .....	219

14.1.7.6 В-дерево .....	220
14.1.7.7 В+ дерево .....	222
14.1.7.8 Красно-чёрное дерево .....	222
14.1.7.9 R-дерево .....	224
14.1.7.10 2-3 деревья .....	224
14.1.7.11 Двоичная куча .....	224
14.1.8 Префиксное дерево .....	225
14.1.9 Hash map .....	226
14.2 Алгоритмы обработки данных .....	227
14.3 Алгоритмы поиска данных .....	227
14.4 Сжатия данных .....	227
14.4.1 Алгоритм Хаффмана .....	228
14.4.2 ZIP (LZ77) .....	231
14.4.2.1 Кодирование.....	232
14.4.2.2 Декодирование .....	235
14.4.3 LZ78.....	237
14.4.3.1 Кодирование.....	237
14.4.3.2 Декодирование .....	239
14.4.4 Сжатие способом кодирования серий .....	239
15 .....	240
15.1 Реляционная модель .....	240
15.1.1 Тип данных .....	244
15.1.2 Атрибут .....	244
15.1.3 Домен атрибута .....	245
15.1.4 Взаимосвязи.....	245
15.1.5 Отношение .....	245
15.1.6 Кортеж.....	245
15.1.7 Схема отношения .....	247
15.1.8 Проекция.....	247
15.2 Реляционная алгебра.....	247
15.2.1 Хороший сайт .....	247
15.2.1.1 Операция выборки .....	248

15.2.1.2 Операция проекции .....	249
15.2.1.3 Операция объединения .....	250
15.2.1.4 Операция пересечения .....	251
15.2.1.5 Операция разности .....	252
15.2.1.6 Операция декартова произведения .....	253
15.2.1.7 Операция деления .....	254
15.2.1.1 Операция соединения.....	254
15.2.2 Книга .....	255
15.2.2.1 Объединение .....	256
15.2.2.2 Пересечение.....	257
15.2.2.3 Разность .....	258
15.2.2.4 Произведение .....	259
15.2.2.5 Сокращение (Выборка, Ограничение).....	260
15.2.2.6 Проекция.....	260
15.2.2.7 Соединение.....	261
15.2.2.8 Деление .....	261
15.2.3 Далее из веба, другое объяснение .....	262
15.2.3.1 Проекция.....	263
15.2.3.2 Выборка .....	264
15.2.3.3 Умножение (Декартово произведение) .....	265
15.2.3.4 Соединение.....	266
15.2.3.5 Пересечение и вычитание.....	268
15.3 Нормальные формы отношений .....	268
15.3.1 Термины .....	269
15.3.2 Первая нормальная форма .....	269
15.3.3 Вторая нормальная форма .....	270
15.3.4 Третья нормальная форма .....	271
15.3.5 Нормальная форма Бойса-Кодда (НФБК) (частная форма третьей нормальной формы).....	272
15.3.6 Четвертая нормальная форма .....	274
15.3.7 Пятая нормальная форма .....	275
15.3.8 Размышления из ютуба .....	276
16 .....	277

16.1 Структуры физического уровня баз данных .....	277
16.1.1 Файлы.....	277
16.1.2 Страницы .....	278
16.1.3 Алгоритмы.....	279
16.2 Методы индексирования .....	280
16.2.1 Про БД.....	280
16.2.1.1 В дерево .....	281
16.2.1.2 Hash индексы.....	281
16.2.1.3 GiST индексы .....	282
16.2.1.4 GIN (инвертированный).....	282
16.2.1.5 Функциональный индекс .....	283
16.2.1.6 Кластерный индекс.....	283
16.2.2 Особенность SQL.....	283
17 .....	284
17.1 Компоненты систем управления базами данных.....	284
17.1.1 Основные функции: .....	284
17.1.2 Состав СУБД .....	284
17.2 Целостность данных.....	285
17.3 Транзакции.....	285
17.3.1 Атомарность (Журнал транзакций) .....	287
17.3.2 Согласованность .....	289
17.3.3 Изолированность.....	289
17.3.3.1 MVCC.....	289
17.3.3.2 Уровни изолированности транзакций .....	294
17.3.4 Долговечность .....	296
18 .....	297
18.1 Архитектура систем баз данных.....	297
18.2 Независимость данных .....	298
18.3 Целостность данных .....	298
18.4 Не избыточное хранение данных .....	300
19 .....	300
19.1 Язык SQL. Средства описания данных, определения ограничений целостности.....	300

20 .....	300
20.1 Язык SQL. Средства манипулирования данными. ....	300
21 .....	300
21.1 Язык XML. ....	300
21.1.1 Xml vs Json.....	301
21.2 Структурная модель документа (DTD).....	301
21.3 Адресация содержания XML-документов согласно спецификации Xlink/Xpointer/XPath. ....	301

# 1

## **1.1 Понятие архитектуры вычислительных систем (ВС).**

Вычислительная машина – это комплекс технических и программных средств, предназначенных для автоматизации и решения задач пользователя.

Вычислительная система – это совокупность взаимосвязанных и взаимосоединенных процессоров или вычислительных машин, периферийного оборудования и программного обеспечения для решения задач пользователя. *это совокупность вычислительных машин, периферийного оборудования и ПО для решения задач пользователя* Под вычислительной системой (ВС) будем понимать совокупность взаимосвязанных и взаимодействующих процессоров или ЭВМ, периферийного оборудования и программного обеспечения, предназначенную для подготовки и решения задач пользователей. Отличительной особенностью ВС по отношению к ЭВМ является наличие в них нескольких вычислителей, реализующих параллельную обработку.

Создание ВС преследует следующие основные цели:

- повышение производительности системы за счет ускорения процессов обработки данных;
- повышение надежности и достоверности вычислений;

– предоставление пользователям дополнительных сервисных услуг.

Параллелизм в вычислениях в значительной степени усложняет управление вычислительным процессом, использование технических и программных ресурсов. Эти функции выполняет операционная система ВС.

Основной отличительной чертой вычислительных систем является наличие в них средств, реализующих параллельную обработку за счет построения параллельных ветвей вычисления, что как правило не предусматривается в вычислительных машинах.

Различия между вычислительными машинами и вычислительными системами не могут быть точно определены (вычислительные машины даже с одним процессором обладает разными средствами распараллеливания, а вычислительные системы могут состоять из традиционных вычислительных машин или процессоров.

Архитектура ВС – совокупность характеристик и параметров, определяющих функционально-логичную и структурно-организованную систему и затрагивающих в основном уровень параллельно работающих вычислителей. Понятие архитектуры охватывает общие принципы построения и функционирования, наиболее существенные для пользователя

Основные принципы построения, закладываемые при создании ВС:

- возможность работы в разных режимах;
- модульность структуры технических и программных средств;
- унификация и стандартизация технических и программных решений;
- иерархия в организации управления процессами;
- способность систем к адаптации, самонастройке и самоорганизации;

## **1.2 Классификация ВС.**

### *1.2.1 По назначению*

Вычислительные системы делят на

- Универсальные
- Специализированные.

Универсальные ВС предназначаются для решения самых различных задач.

Специализированные системы ориентированы на решение узкого класса задач.

Специализация ВС может устанавливаться различными средствами:

- Структурой системы (количество параллельно работающих элементов, связи между ними и т.д.) может быть ориентирована на определенные виды обработки информации: матричные вычисления, решение алгебраических, дифференциальных и интегральных уравнений и т.п. Практика разработки ВС типа супер ЭВМ показала, чем выше их производительность, тем уже класс эффективно решаемых ими задач;

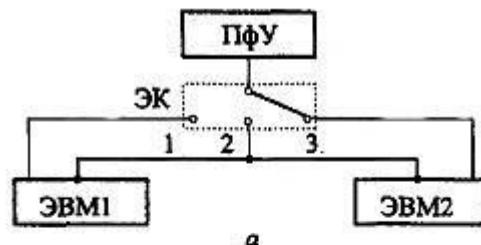
- Специализация ВС может закладываться включением в их состав специального оборудования и специальных пакетов обслуживания техники.

### 1.2.2 По типу

Вычислительные системы различаются на

- многомашинные
- многопроцессорные.

Многомашинные вычислительные системы (MMC) появились исторически первыми. Уже при использовании ЭВМ первых поколений возникали задачи повышения производительности, надежности и достоверности вычислений. Для этих целей использовали комплекс машин, схематически показанный на рис. а.



Положения 1 и 3 электронного ключа (ЭК) обеспечивало режим повышенной надежности. При этом одна из машин выполняла вычисления, а другая находилась в “горячем” или “холодном” резерве, т.е. в готовности заменить основную ЭВМ. Положение 2 электронного ключа соответствовало случаю, когда обе машины обеспечивали параллельный режим вычислений.

Здесь возможны две ситуации:

- обе машины решают одну и ту же задачу и периодически сверяют результаты, решения. Тем самым обеспечивался режим повышенной достоверности, уменьшалась вероятность появления ошибок в результатах вычислений. Примерно по такой же схеме построены

управляющие бортовые вычислительные комплексы космических аппаратов, ракет, кораблей. Например, американская космическая система “Шатл” содержала пять вычислительных машин, работающих по такой схеме;

- обе машины работают параллельно, но обрабатывают собственные потоки заданий. Возможность обмена информацией между машинами сохраняется. Этот вид работы относится к режиму повышенной производительности. Она широко используется в практике организации работ на крупных вычислительных центрах, оснащенных несколькими ЭВМ высокой производительности.

Схема, представленная на рис. а, была неоднократно повторена в различных модификациях при проектировании разнообразных специализированных ММС. Основные различия ММС заключаются, как правило, в организации связи и обмена информацией между ЭВМ комплекса. Каждая из них сохраняет возможность автономной работы и управляет собственной ОС. Любая другая подключаемая ЭВМ комплекса рассматривается как специальное периферийное оборудование.

Многопроцессорные вычислительные системы (МПС) строятся при комплексировании нескольких процессоров (рис. б). В качестве общего ресурса они имеют общую оперативную память (ООП). Параллельная работа процессоров и использование ООП обеспечиваются под управлением единой общей операционной системы. По сравнению с ММС здесь достигается наивысшая оперативность взаимодействия вычислителей-процессоров. Многие исследователи считают, что использование МПС является основным магистральным путем развития вычислительной техники новых поколений.

Однако МПС имеют и существенные недостатки. Они в первую очередь связаны с использованием ресурсов общей оперативной памяти. При большом количестве комплексируемых процессоров возможно возникновение конфликтных ситуаций, когда несколько процессоров обращаются с операциями типа “чтение” и “запись” к одним и тем же областям памяти. Помимо процессоров к ООП подключаются все каналы (процессоры ввода-вывода), средства измерения времени и т.д. Поэтому вторым серьезным недостатком МПС является проблема коммутации абонентов и доступа их к ООП. От того, насколько удачно решаются эти проблемы, и зависит эффективность применения МПС. Это решение обеспечивается аппаратурно-программными

средствами. Процедуры взаимодействия очень сильно усложняют структуру ОС МПС.

### *1.2.3 По типу ЭВМ или процессоров*

Используемых для построения ВС, различают

- однородные
- неоднородные системы.

Однородные системы предполагают комплексирование однотипных ЭВМ (процессоров), неоднородные - разнотипных. В однородных системах значительно упрощаются разработка и обслуживание технических и программных (в основном ОС) средств. В них обеспечивается возможность стандартизации и унификации соединений и процедур взаимодействия элементов системы. Упрощается обслуживание систем, облегчаются модернизация и их развитие. Вместе с тем существуют и неоднородные ВС, в которых комплексируемые элементы очень сильно отличаются по своим техническим и функциональным характеристикам. Обычно это связано с необходимостью параллельного выполнения многофункциональной обработки. Так, при построении ММС, обслуживающих каналы связи, целесообразно объединять в комплекс связанные, коммуникационные машины и машины обработки данных. В таких системах коммуникационные ЭВМ выполняют функции связи, контроля получаемой и передаваемой информации, формирования пакетов задач и т.д. ЭВМ обработки данных не занимаются не свойственными им работами по обеспечению взаимодействия в сети, а все их ресурсы переключаются на обработку данных. Неоднородные системы находят применение и в МПС.

Многие ЭВМ, в том числе и ПЭВМ, могут использовать сопроцессоры: десятичной арифметики, матричные и т.п.

### *1.2.4 По степени территориальной разобщенности*

Вычислительных модулей ВС делятся на системы

- совмещенного (сосредоточенного)
- распределенного (разобщенного) типов.

Обычно такое деление касается только ММС. Многопроцессорные системы относятся к системам совмещенного типа. Более того, учитывая успехи микроэлектроники, это совмещение может быть очень глубоким. При появлении

новых СБИС (сверхбольших интегральных схем) появляется возможность иметь в одном кристалле несколько параллельно работающих процессоров.

Совмещенные и распределенные ММС сильно отличаются оперативностью взаимодействия в зависимости от удаленности ЭВМ. Время передачи информации между соседними ЭВМ, соединенными простым кабелем, может быть много меньше времени передачи данных по каналам связи. Как правило, все выпускаемые в мире ЭВМ имеют средства прямого взаимодействия и средства подключения к сетям ЭВМ. Для ПЭВМ такими средствами являются нуль-модемы, модемы и сетевые карты как элементы техники связи.

По методам управления элементами ВС различают централизованные, децентрализованные и со смешанным управлением. Помимо параллельных вычислений, производимых элементами системы, необходимо выделять ресурсы на обеспечение управления этими вычислениями. В централизованных ВС за это отвечает главная, или диспетчерская, ЭВМ (процессор). Ее задачей являются распределение нагрузки между элементами, выделение ресурсов, контроль состояния ресурсов, координация взаимодействия. Централизованный орган управления в системе может быть жестко фиксирован или эти функции могут передаваться другой ЭВМ (процессору), что способствует повышению надежности системы. Централизованные системы имеют более простые ОС. В децентрализованных системах функции управления распределены между ее элементами. Каждая ЭВМ (процессор) системы сохраняет известную автономию, а необходимое взаимодействие между элементами устанавливается по специальным наборам сигналов. С развитием ВС и, в частности, сетей ЭВМ интерес к децентрализованным системам постоянно растет.

В системах со смешанным управлением совмещаются процедуры централизованного и децентрализованного управления. Перераспределение функций осуществляется в ходе вычислительного процесса исходя из сложившейся ситуации.

По принципу закрепления вычислительных функций за отдельными ЭВМ (процессорами) различают системы с жестким и плавающим закреплением функций. В зависимости от типа ВС следует решать задачи статического или динамического размещения программных модулей и массивов данных, обеспечивая необходимую гибкость системы и надежность ее функционирования.

По режиму работы ВС различают системы, работающие в оперативном и неоперативном временных режимах. Первые, как правило, используют режим

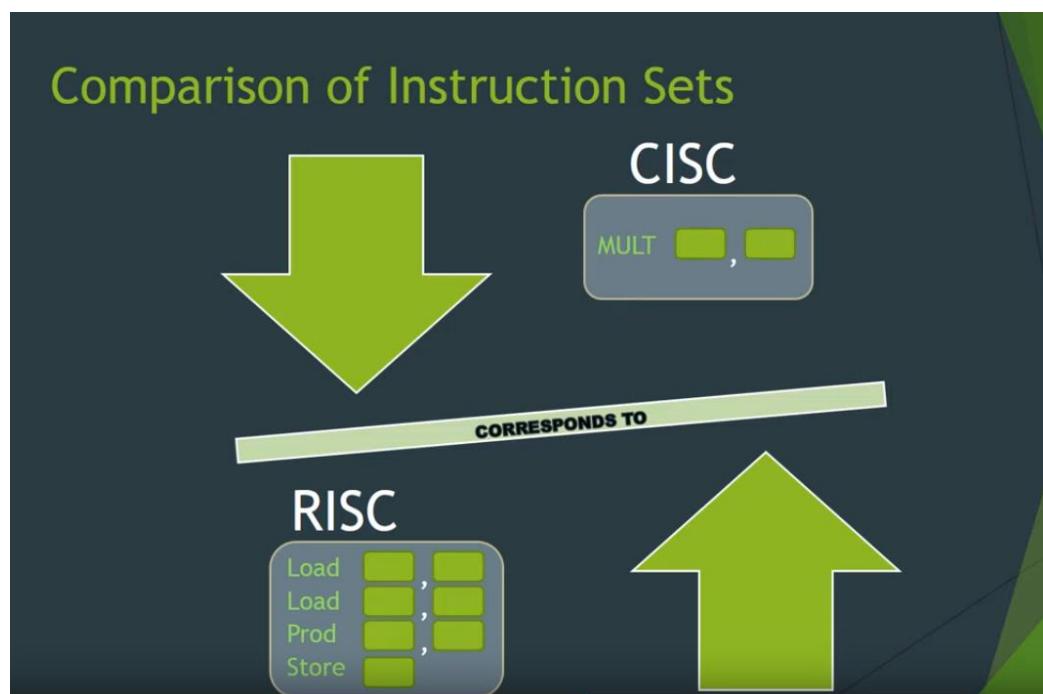
реального масштаба времени. Этот режим характеризуется жесткими ограничениями на время решения задач в системе и предполагает высокую степень автоматизации процедур ввода-вывода и обработки данных.

Наибольший интерес у исследователей всех рангов (проектировщиков, аналитиков и пользователей) вызывают структурные признаки ВС. От того, насколько структура ВС соответствует структуре решаемых на этой системе задач, зависит эффективность применения ЭВМ в целом. Структурные признаки, в свою очередь, отличаются многообразием: топология управляющих и информационных связей между элементами системы, способность системы к перестройке и перераспределению функций, иерархия уровней взаимодействия элементов. В наибольшей степени структурные характеристики определяются архитектурой системы.

### 1.3 Принципы организации CISC и RISC архитектур.

#### 1.3.1 Вместе

Раньше память была дорога, поэтому чем меньше код - тем лучше, тк в CISC это была бы лишь одна строчка. RISC жрет меньше питания.



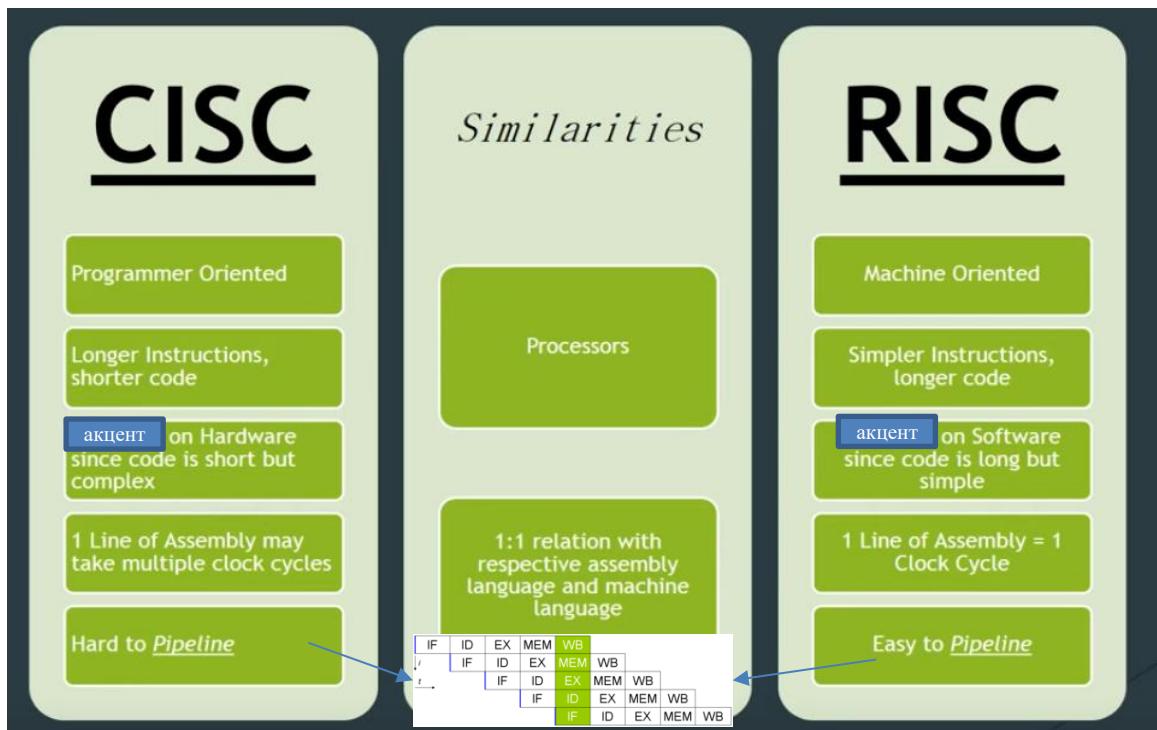


Рисунок 1

Конвейер (**Pipeline**) — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств.

### 1.3.2 RISC

RISC (Reduced Instruction Set Computing). Процессор с сокращенным набором команд. Система команд имеет упрощенный вид. Все команды одинакового формата с простой кодировкой. Обращение к памяти происходит посредством команд загрузки и записи, остальные команды типа регистр-регистр. Команда, поступающая в CPU, уже разделена по полям и не требует дополнительной дешифрации.

Часть кристалла освобождается для включения дополнительных компонентов. Степень интеграции ниже, чем в предыдущем архитектурном варианте, поэтому при высоком быстродействии допускается более низкая тактовая частота. Команда меньше загромождает ОЗУ, CPU дешевле. Программной совместимостью указанные архитектуры не обладают. Отладка

программ на RISC более сложна. Данная технология может быть реализована программно-совместимым с технологией CISC (например, суперскалярная технология).

Поскольку RISC-инструкции просты, для их выполнения нужно меньше логических элементов, что в конечном итоге снижает стоимость процессора. Но большая часть программного обеспечения сегодня написана и откомпилирована специально для CISC-процессоров фирмы Intel. Для использования архитектуры RISC нынешние программы должны быть перекомпилированы, а иногда и переписаны заново.

Достоинства архитектуры RISC:

- снижение нерегулярности потока команд
- обогащение пространственным параллелизмом

Недостатки архитектуры RISC:

RISC (англ. reduced instruction set computer — компьютер с набором коротких (простых, быстрых) команд) — архитектура процессора, в которой быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим. Первые RISC-процессоры даже не имели инструкций умножения и деления. Это также облегчает повышение тактовой частоты и делает более эффективной суперскалярность (распараллеливание инструкций между несколькими исполнительными блоками).

### 1.3.3 CISC

Эта архитектура имеет специализированный набор команд. CISC (англ. Complex Instruction Set Computing) — концепция проектирования процессоров, которая характеризуется следующим набором свойств:

- большим числом различных по формату и длине команд;
- введением большого числа различных режимов адресации;
- обладает сложной кодировкой инструкций.

Процессору с архитектурой CISC приходится иметь дело с более сложными инструкциями неодинаковой длины. Выполнение одиночной CISC-инструкции может происходить быстрее, однако обрабатывать несколько таких инструкций параллельно сложнее.

Облегчение отладки программ на ассемблере влечет за собой загромождение узлами микропроцессорного блока. Для повышения быстродействия следует увеличить тактовую частоту и степень интеграции, что

вызывает необходимость совершенствования технологии и, как следствие, более дорогостоящего производства.

Достоинства архитектуры CISC:

- Компактность наборов инструкций уменьшает размер программ и уменьшает количество обращений к памяти.
- Наборы инструкций включают поддержку конструкций высокоуровневого программирования.

Недостатки архитектуры CISC:

- Нерегулярность потока команд.
- Высокая стоимость аппаратной части.
- Сложности с распараллеливанием вычислений.

#### *1.3.4 MISC*

MISC (Multipurpose Instruction Set Computer). Элементная база состоит из двух частей, которые либо выполнены в отдельных корпусах, либо объединены. Основная часть – RISC CPU, расширяемый подключением второй части – ПЗУ микропрограммного управления. Система приобретает свойства CISC. Основные команды работают на RISC CPU, а команды расширения преобразуются в адрес микропрограммы. RISC CPU выполняет все команды за один такт, а вторая часть эквивалентна CPU со сложным набором команд. Наличие ПЗУ устраняет недостаток RISC, выраженный в том, что при компиляции с языка высокого уровня микрокод генерируется из библиотеки стандартных функций, занимающей много места в ОЗУ. Поскольку микропрограмма уже дешифрована и открыта для программиста, то времени выборки из ОЗУ на дешифрацию не требуется.

## 2

### **2.1 Многопроцессорные системы. Симметричная и асимметричная многопроцессорность.**

#### *2.1.1 Симметричная многопроцессорность*

Термин SMP (SMP, Symmetric Multiprocessor) относится как к архитектуре ВС, так и к поведению операционной системы, отражающему данную архитектурную организацию. SMP можно определить как вычислительную систему, обладающую следующими характеристиками:

- имеются два или более процессоров сопоставимой производительности;
- процессоры совместно используют основную память и работают в едином виртуальном и физическом адресном пространстве;
- все процессоры связаны между собой посредством шины или по иной схеме, так что время доступа к памяти любого из них одинаково;
- все процессоры разделяют доступ к устройствам ввода/вывода либо через один и тот же каналы, либо через разные каналы, обеспечивающие доступ к одному и тому же внешнему устройству;
- все процессоры способны выполнять одинаковые функции (этим объясняется термин «симметричные»);
- любой из процессоров может обслуживать внешние прерывания;
- вычислительная система управляет интегрированной операционной системой, которая организует и координирует взаимодействие между процессорами и программами на уровне заданий, задач, файлов и элементов данных.

Обратим внимание на последний пункт, который подчеркивает одно из отличий по отношению к слабо связанным мультипроцессорным системам, таким как кластеры, где в качестве физической единицы обмена информацией обычно выступает сообщение или полный файл. В SMP допустимо взаимодействие на уровне отдельного элемента данных, благодаря чему может быть достигнута высокая степень связности между процессами. Хотя технически SMP-системы симметричны, в их работе присутствует небольшой фактор перекоса, который вносит программное обеспечение. На время

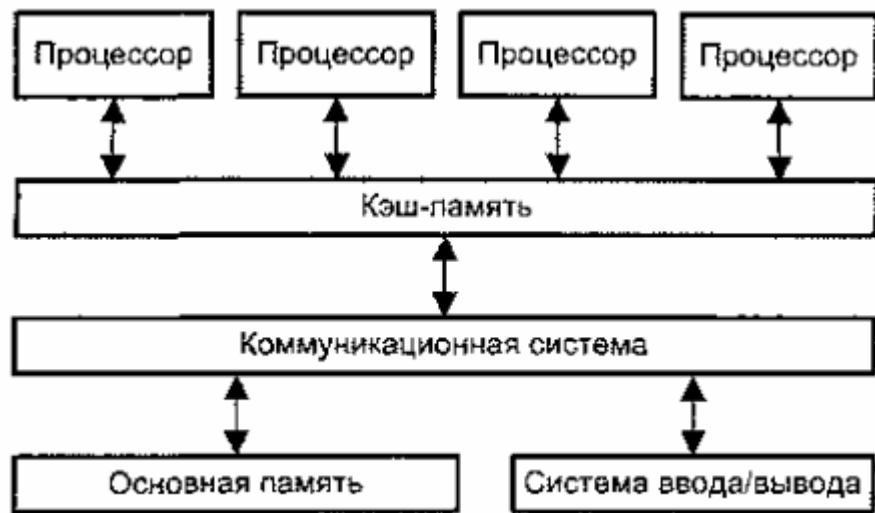
загрузки системы один из процессоров получает статус ведущего (master). Это не означает, что позже, во время работы какие-то процессоры будут ведомыми - все они в SMP-системе равноправны. Термин «ведущий» вводится только затем, чтобы указать, какой из процессоров по умолчанию будет руководить первоначальной загрузкой ВС. Операционная система планирует процессы или нити процессов (threads) сразу для всех процессоров, скрывая при этом от пользователя многопроцессорный характер SMP-архитектуры. На рис. 2.27 в самом общем виде показана архитектура симметричной мультипроцессорной ВС. Типовая SMP-система содержит от двух до 32 идентичных процессоров, в качестве которых обычно выступают недорогие RISC-процессоры, такие, например, как DEC Alpha, Sun SPARC, MIPS или HP

PA-RISC. В последнее время наметилась тенденция оснащения SMP-системами также и CISC-процессорами, в частности Pentium или AMD



Каждый процессор снабжен локальной кэш-памятью, состоящей из кэш-памяти первого (L1) и второго (L2) уровней. Согласованность содержимого кэш-памяти всех процессоров обеспечивается аппаратными средствами. В некоторых SMP-системах (рис. 2.28). К сожалению, этот прием технически и экономически оправдан лишь,

если число процессоров не превышает четырех. Применение общей кэш-памяти сопровождается повышением стоимости и снижением быстродействия кэш-памяти. Все процессоры ВС имеют равноправный доступ к разделяемым основной памяти и устройствам ввода/вывода. Такая возможность обеспечивается коммуникационной системой. Обычно процессоры взаимодействуют между собой через основную память (сообщения и информация о состоянии оставляются в области общих данных). В некоторых SMP-системах предусматривается также прямой обмен сигналами между процессорами. Память системы обычно строится по модульному принципу и организована так, что допускается одновременное обращение к разным сеансам модулям (банкам). В некоторых конфигурациях в дополнение к совместно используемым ресурсам каждый процессор обладает также собственными локальной основной памятью и каналами ввода/вывода.

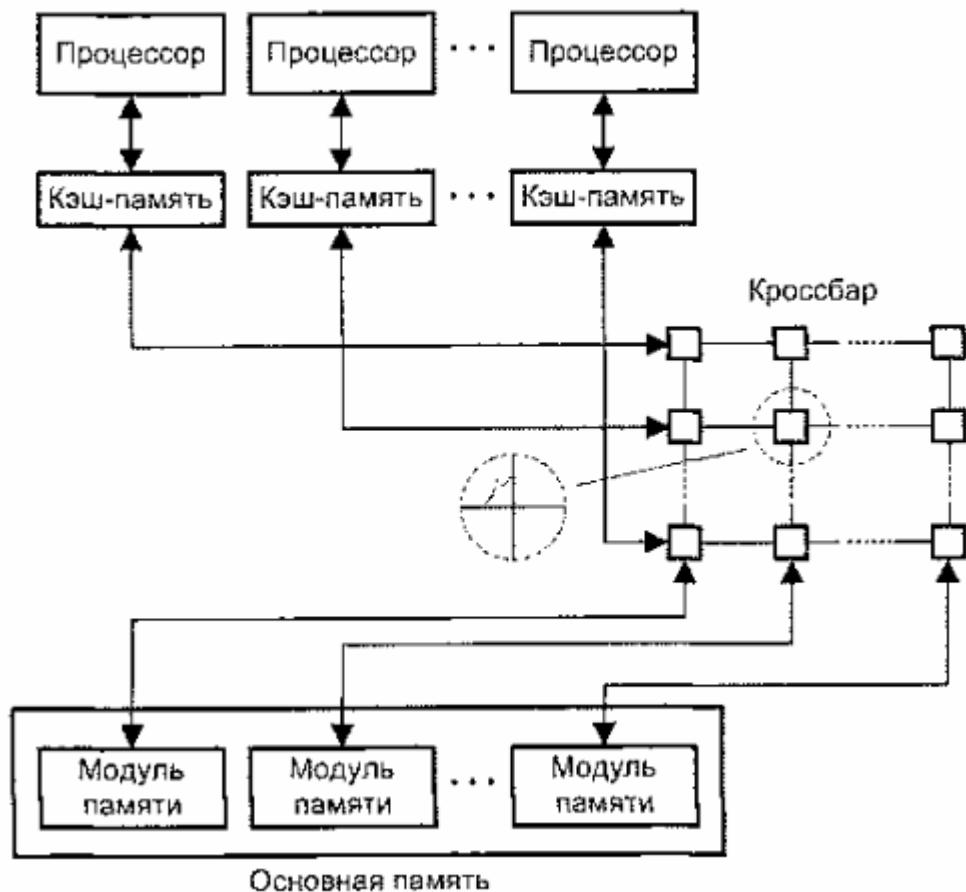


Важным аспектом архитектуры симметричных мультипроцессоров является способ взаимодействия процессоров с общими ресурсами (памятью и системой ввода/вывода). С этих позиций можно выделить следующие виды архитектуры SMP-систем: с общей шиной и временным разделением; с коммутатором типа «кроссбар»; с многопортовой памятью; с централизованным устройством управления. Структура и интерфейсы общей шины в основном такие же, как и в однопроцессорной ВС, где шина служит для внутренних соединений (рис. 2.29). Достоинства и недостатки систем коммуникации на базе общей шины с разделением времени достаточно подробно обсуждались ранее. Применительно к SMP-

системам отметим, что физический интерфейс, а также логика адресации, арбитража и разделения времени остаются теми же, что и в однопроцессорных системах.



Общая шина позволяет легко расширять систему путем подключения к себе большего числа процессоров. Кроме того, напомним, что шина — это, по существу, пассивная среда, и отказ одного из подключенных к ней устройств не влечет отказа всей совокупности. В то же время SMP-системам на базе общей шины свойственен и основной недостаток шинной организации — невысокая производительность: скорость системы ограничена временем цикла шины. По этой причине каждый процессор снабжен кэш-памятью, что существенно уменьшает число обращений к шине. Наличие множества кэшей порождает проблему их когерентности, и это одна из основных причин, по которой системы на базе общей шины обычно содержат не слишком много процессоров. Так, в системах Compaq AlphaServer GS140 и 8400 используется не более 14 процессоров Alpha 21264. SMP-система HP N9000 в максимальном варианте состоит из 8 процессоров PA-8500, а система SMP Thin Nodes для RS/6000 фирмы IBM может включать в себя от двух до четырех процессоров PowerPC 604. Архитектура с коммутатором типа «кроссбар» (рис. 2.30) ориентирована на модульное построение общей памяти и призвана разрешить проблему ограниченной пропускной способности систем с общей шиной. Коммутатор обеспечивает множественность путей между процессорами и банками памяти, причем топология связей может быть как двумерной, так и трехмерной. Результатом становится более высокая полоса пропускания, что позволяет строить SMP-системы, содержащие больше процессоров, чем в случае общей шин. Типичное число процессоров в SMP-системах на базе матричного коммутатора составляет 32 или 64. Отметим, что выигрыш в производительности достигается, лишь когда разные процессоры обращаются к разным банкам памяти.



По логике кроссбара строится и взаимодействие процессоров с устройствами ввода/вывода. В качестве примера ВС с рассмотренной архитектурой можно привести систему Enterprise 10000, состоящую из 64 процессоров, связанных с памятью посредством матричного коммутатора Gigaplane-XB фирмы Sun Microsystems (кроссбар 1G x 16). В IBM RS/6000 Enterprise Server Model S70 коммутатор типа «кроссбар» обеспечивает работу 12 процессоров RS64. В SMP-системах ProLiant 8000 и 8500 фирмы Compaq для объединения с памятью и между собой восьми процессоров Pentium III Xeon применена комбинация нескольких шин и кроссбара. Концепция матричного коммутатора (кроссбара) не ограничивается симметричными мультипроцессорами. Аналогичная структура связей применяется для объединения узлов в ВС типа CC-NUMA и кластерных вычислительных системах.

Многопортовая организация запоминающего устройства обеспечивает любому процессору и модулю ввода/вывода прямой и непосредственный доступ к банкам основной памяти (ОП). Такой подход сложнее, чем при использовании шины, поскольку требует придания ЗУ основной памяти дополнительной, достаточно сложной логики. Тем не менее это позволяет поднять

производительность, так как каждый процессор имеет выделенный тракт к каждому модулю ОП. Другое преимущество многопортовой организации - возможность назначить отдельные модули памяти в качестве локальной памяти отдельного процессора. Эта особенность позволяет улучшить защиту данных от несанкционированного доступа со стороны других процессоров. Централизованное устройство управления (ЦУУ) сводит вместе отдельные потоки данных между независимыми модулями: процессором, памятью, устройствами ввода/вывода. ЦУУ может буферизировать запросы, выполнять синхронизацию и арбитраж. Оно способно передавать между процессорами информацию о состоянии и управляющие сообщения, а также предупреждать об изменении информации в кэшах. Недостаток такой организации заключается в сложности устройства управления, что становится потенциальным узким местом в плане производительности. В настоящее время подобная архитектура встречается достаточно редко, но она широко использовалась при создании вычислительных систем на базе машин семейства IBM 370.

#### *2.1.1.1 Википедия*

Симметрическая многопроцессорность (англ. Symmetric Multiprocessing, сокращённо SMP) — архитектура многопроцессорных компьютеров, в которой два или более одинаковых процессора сравнимой производительности подключаются единообразно к общей памяти (и периферийным устройствам) и выполняют одни и те же функции (почему, собственно, система и называется симметрической). В английском языке SMP-системы носят также название *tightly coupled multiprocessors*, так как в этом классе систем процессоры тесно связаны друг с другом через общую шину и имеют равный доступ ко всем ресурсам вычислительной системы (памяти и устройствам ввода-вывода) и управляются все одной копией операционной системы. Большинство многопроцессорных систем сегодня используют архитектуру SMP.

#### *2.1.1.2 Описание*

SMP-системы позволяют любому процессору работать над любой задачей независимо от того, где в памяти хранятся данные для этой задачи, — при должной поддержке операционной системой SMP-системы могут легко перемещать задачи между процессорами, эффективно распределяя нагрузку.

Разные SMP-системы соединяют процессоры с общей памятью по-разному. Самый простой и дешевый подход — это соединение по общей шине (system bus)[3][4]. В этом случае только один процессор может обращаться к

памяти в каждый данный момент, что накладывает существенное ограничение на количество процессоров, поддерживаемых в таких системах. Чем больше процессоров, тем больше нагрузка на общую шину, тем дольше должен ждать каждый процессор, пока освободится шина, чтобы обратиться к памяти. Снижение общей производительности такой системы с ростом количества процессоров происходит очень быстро, поэтому обычно в таких системах количество процессоров не превышает 2-4. Примером SMP-машин с таким способом соединения процессоров являются любые многопроцессорные серверы начального уровня.

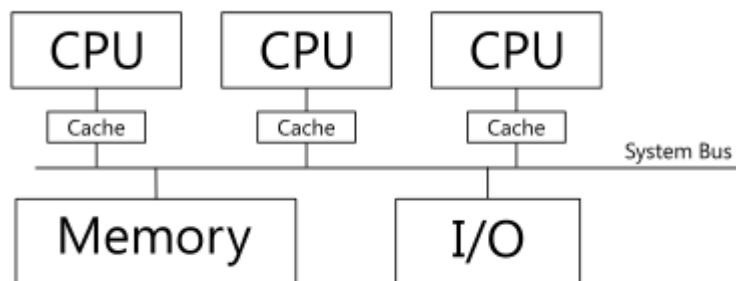


Рисунок 2

Второй способ соединения процессоров — через коммутируемое соединение (crossbar switch). При таком соединении вся общая память делится на банки памяти, каждый банк памяти имеет свою собственную шину, и процессоры соединены со всеми шинами, имея доступ по ним к любому из банков памяти. Такое соединение схемотехнически более сложное, но оно позволяет процессорам обращаться к общей памяти одновременно. Это позволяет увеличить количество процессоров в системе до 8-16 без заметного снижения общей производительности. Примером таких SMP-машин являются многопроцессорные рабочие станции RS/6000.

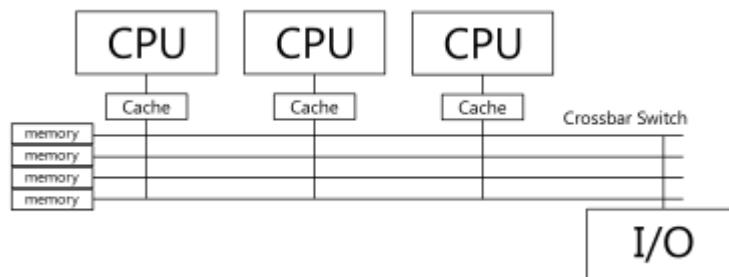


Рисунок 3

### *2.1.1.3 Преимущества и недостатки*

SMP — самый простой и экономически выгодный способ масштабирования вычислительной системы: путём наращивания числа процессоров. Также просто и программирование: с помощью потоков и сопутствующих механизмов обмена данными между ними через общие переменные в памяти.

SMP часто применяется в науке, промышленности, бизнесе, где программное обеспечение специально разрабатывается для многопоточного выполнения. В то же время большинство потребительских продуктов, таких как текстовые редакторы и компьютерные игры, написаны так, что они не могут использовать сильные стороны SMP-систем. В случае игр это зачастую связано с тем, что оптимизация программы под SMP-системы приведёт к потере производительности при работе на однопроцессорных системах, которые еще недавно занимали большую часть рынка ПК. (Современные многоядерные процессоры — лишь еще одна аппаратная реализация SMP.) В силу природы разных методов программирования для максимальной производительности потребуются отдельные проекты для поддержки одного одноядерного процессора и SMP-систем. И все же программы, запущенные на SMP-системах, получают незначительный прирост производительности, даже если они были написаны для однопроцессорных систем. Это связано с тем, что аппаратные прерывания, обычно приостанавливающие выполнение программы для их обработки ядром, могут обрабатываться на свободном процессоре (процессорном ядре). Эффект в большинстве приложений проявляется не столько в приросте производительности, сколько в ощущении, что программа выполняется более плавно. В некоторых прикладных программах (в частности: программных компиляторах и некоторых проектах распределённых вычислений) повышение производительности будет почти прямо пропорционально числу дополнительных процессоров.

Выход из строя одного процессора приводит к некорректной работе всей системы и требует перезагрузки всей системы для отключения неисправного процессора. Выход же из строя одного процессорного ядра нередко оборачивается выходом из строя всего многоядерного процессора, если многоядерный процессор не оснащен встроенной защитой, отключающей неисправное процессорное ядро и через это позволяющей исправным процессорным ядрам продолжать работать штатно.

В случае отказа одного из процессоров симметричные системы, как правило, сравнительно просто реконфигурируются, что является их большим преимуществом перед плохо реконфигурируемыми асимметричными системами.

#### *2.1.1.4 Ограничение на количество процессоров*

При увеличении числа процессоров заметно увеличивается требование к пропускной возможности шины памяти. Это налагает ограничение на количество процессоров в SMP архитектуре. Современные SMP-системы позволяют эффективно работать при 16 процессорах.

#### *2.1.1.5 Проблема когерентности кэш-памяти*

Каждый современный процессор оборудован многоуровневой кэш-памятью для более быстрой выборки данных и машинных команд из основной памяти, которая работает медленнее, чем процессор. В многопроцессорной системе наличие кэш-памяти у процессоров снижает нагрузку на общую шину или на коммутируемое соединение, что весьма благоприятно сказывается на общей производительности системы. Но так как каждый процессор оборудован своей индивидуальной кэш-памятью, то возникает опасность, что в кэш-память одного процессора попадёт значение переменной, отличное от того, что хранится в основной памяти и в кэш-памяти другого процессора. Представим, что процессор изменяет значение переменной в своем кэше, а другой процессор запрашивает эту переменную из основной памяти, — и он (второй процессор) получит уже недействительное значение переменной. Или, например, подсистема ввода-вывода записывает в основную память новое значение переменной, а в кэш-памяти процессора все еще остается устаревшее. Разрешение этой проблемы возложено на протокол согласования кэшей (cache coherence protocol), который призван обеспечить согласованность («когерентность») кэшей всех процессоров и основной памяти без потери общей производительности.

#### *2.1.1.6 Поддержка операционной системой*

Поддержка SMP должна быть встроена в операционную систему, иначе дополнительные процессоры будут бездействовать, и система будет работать как однопроцессорная. (Собственно эта проблема актуальна и для однопроцессорных систем с многоядерным процессором.) Большинство

современных операционных систем поддерживает симметричную мультипроцессорность, но в разной степени.

Поддержка многопроцессорности в ОС Linux была добавлена в версии ядра 2.0[7] и усовершенствована в версии 2.6. Линейка ОС Windows NT изначально создавалась с поддержкой многопроцессорности. (Windows 9x SMP не поддерживали.)

### *2.1.1.7 Альтернативы*

SMP — лишь один вариант построения многопроцессорной машины. Другая концепция — NUMA, которая предоставляет процессорам отдельные банки памяти. Это позволяет процессорам работать с памятью параллельно, и это может значительно повысить пропускную способность памяти в случае когда данные привязаны к конкретному процессу (а, следовательно, и процессору). С другой стороны, NUMA повышает стоимость перемещения данных между процессорами, значит, и балансирование загрузки обходится дороже. Преимущества NUMA ограничены специфическим кругом задач, в основном серверами, где данные часто жестко привязаны к конкретным задачам или пользователям.

Другая концепция — асимметричное мультипроцессорование (ASMP), в котором отдельные специализированные процессоры используются для конкретных задач, и кластерная мультипроцессорность (Beowulf), в которой не вся память доступна всем процессорам. Такие подходы нечасто используются (хотя высокопроизводительные 3D-чипсеты в современных видеокартах могут рассматриваться как разновидность асимметричной мультипроцессорности), в то время как кластерные системы широко применяются при построении очень больших суперкомпьютеров.

### *2.1.2 Асимметричная многопроцессорность*

Это архитектура суперкомпьютера, в которой каждый процессор имеет свою оперативную память. В ASMP используются три схемы (рис. 2.26). В любом случае процессоры взаимодействуют между собой, передавая друг другу сообщения, т. е. как бы образуя скоростную локальную сеть. Передача сообщений может осуществляться через общую шину (рис. 2.26, а см. также MPP-архитектура) либо благодаря межпроцессорным связям. В последнем случае процессоры связаны либо непосредственно (рис. 2.26, б), либо через друг

друга (рис. 2.26, в). Непосредственные связи используются при небольшом числе процессоров.

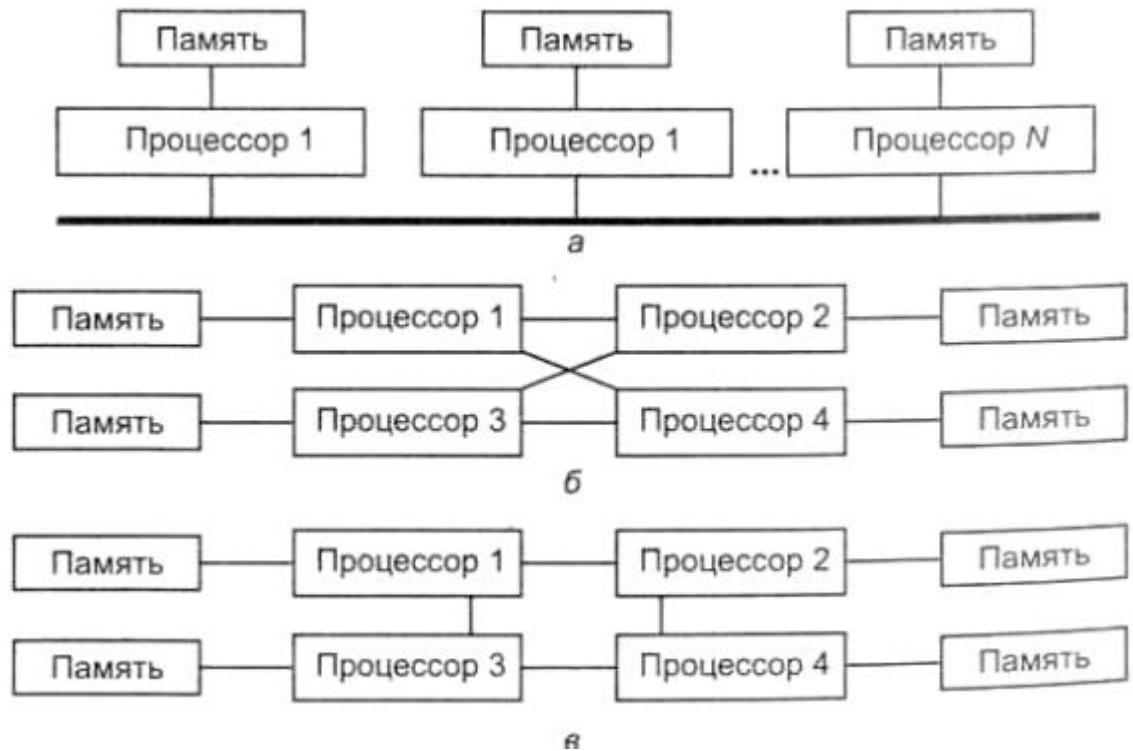


Рисунок 4 2.26

Каждый процессор имеет свою, расположенную рядом с ним оперативную память. Благодаря этому, если это необходимо, процессоры могут располагаться в различных, но рядом установленных корпусах. Совокупность устройств в одном корпусе называется кластером. Пользователь, обращаясь к кластеру, может работать сразу с группой процессоров. Такое объединение увеличивает скорость обработки данных и расширяет используемую оперативную память. Резко возрастает также отказоустойчивость, ибо кластеры осуществляют резервное дублирование данных. Созданная таким образом система называется кластерной. В этой системе, в соответствии с ее структурой, может функционировать несколько копий операционной системы и несколько копий прикладной программы, которые работают с одной и той же базой данных (БД), решая одновременно разные задачи

### 2.1.2.1 Википедия

AMP или ASMP (от англ.: Asymmetric multiprocessing, рус.: Асимметричная многопроцессорная обработка или Асимметричное мультипроцессорование) — тип многопроцессорной архитектуры компьютерной системы, который использовался до того, как была создана

технология симметричного мультипроцессирования (SMP). Также использовался как более дешевая альтернатива в системах, которые поддерживали SMP.

В системе с асимметричной многопроцессорностью не все процессоры играют одинаковую роль. Например, система может использовать (либо на аппаратном, либо на уровне операционной системы) только один процессор для выполнения кода операционной системы, или поручать только одному процессору выполнение операций ввода-вывода. В других AMP-системах все процессоры могут выполнять код операционной системы и операции ввода-вывода, так что с этой стороны они ведут себя как симметричная многопроцессорная система, но определенная периферийная аппаратура может быть подсоединенна только к одному процессору, так что со стороны работы с этой аппаратурой система предстаёт асимметричной.

В 1960-е—1970-е годы увеличить вычислительную мощность компьютера можно было, просто добавив в него ещё один процессор. Добавить ещё один такой же процессор было дешевле, чем покупать новый компьютер, работающий вдвое быстрее. Также простое добавление ещё одного процессора было выгоднее, чем покупка второго целого компьютера, для которого требовался отдельный машинный зал, дополнительное периферийное оборудование и персонал для обслуживания.

Однако проблема с добавлением процессора заключалась в том, что операционные системы того времени были разработаны только для однопроцессорных компьютеров, а внесение изменений для надежной поддержки двух процессоров требовало много времени. Чтобы обойти эту трудность, в операционные системы с поддержкой одного процессора вносили незначительные изменения лишь для минимальной поддержки второго процессора. «Минимальная поддержка» означала, что операционная система запускалась на «загрузочном процессоре» (boot processor), а на втором процессоре исполнялись только пользовательские программы. Например, в Burroughs B5000 второй процессор аппаратно не имел возможности исполнять «управляющий код»[2].

В других системах операционная система могла работать на любом из процессоров, но либо все периферийные устройства присоединялись к какому-то одному процессору, либо через каждый процессор был доступ только к каким-то определенным периферийным устройствам.

Асимметричное мультипроцессирование – наиболее простой способ организации вычислительного процесса в системах с несколькими процессорами. Этот способ часто называют также “ведущий-ведомый”.

Функционирование системы по принципу “ведущий-ведомый” предполагает выделение одного из процессоров в качестве “ведущего”, на котором работает операционная система и который управляет всеми остальными “ведомыми” процессорами, т.е. ведущий процессор берет на себя функции распределения задач и ресурсов, а ведомые процессоры работают только как обрабатывающие устройства и никаких действий по организации работы вычислительной системы не выполняют.

Так как операционная система работает только на одном процессоре и функции управления полностью централизованы, то такая операционная система оказывается не намного сложнее ОС однопроцессорной системы.

## 3

### 3.1 Методы организации сетей ЭВМ

Основными требованиями, которым должна удовлетворять организация сети, являются следующие:

- Открытость - возможность включения дополнительных абонентских, ассоциативных ЭВМ, а также линий (каналов) связи без изменения технических и программных средств существующих компонентов сети. Кроме того, любые две ЭВМ должны взаимодействовать между собой, несмотря на различие в конструкции, производительности, месте изготовления, функциональном назначении.
- Гибкость - сохранение работоспособности при изменении структуры в результате выхода из строя ЭВМ или линии связи.
- Эффективность - обеспечение требуемого качества обслуживания пользователей при минимальных затратах.

Международной организацией стандартов утверждены определённые требования к организации взаимодействия между системами сети. Эти требования получили название OSI (Open System Interconnection) - "эталонная модель взаимодействия открытых систем". Согласно требованиям эталонной модели, каждая система сети должна осуществлять взаимодействие посредством передачи кадра данных. Согласно модели OSI образование и передача кадра

осуществляется с помощью 7-ми последовательных действий, получивших название "уровень обработки".

### **3.2 Основные принципы их функционирования.**

В 1984 году Международная организация по стандартизации ISO (International Standard Organization) закончила начатую в 1977 году разработку модели открытого системного взаимодействия OSI (Open System Interface), которая является в настоящее время международным стандартом для передачи данных по сетям ЭВМ. Модель OSI определяет:

- Способы установки связи и обмена данными между сетевыми устройствами при использовании ими различных систем кодирования данных;
- Методы определения момента начала передачи данных;
- Методы обеспечения передачи нужной информации конкретным адресатам;
- Организацию коммутации элементов физической среды передачи данных;
- Поддержание необходимой скорости передачи данных всеми сетевыми устройствами;
- Методы представления двоичных битов в среде передачи данных.

Модель OSI не описывает ничего реального - это концептуальная основа, с помощью которой общая задача передачи данных разделяется на отдельные легко обозримые компоненты. Модель OSI реализована в виде сетевых протоколов. Под сетевым протоколом понимается соглашение между производителями сетевого оборудования и программного обеспечения о способах обмена информацией между ЭВМ.

### **3.3 Классификация сетей по масштабу и топологии.**

#### *3.3.1 Искусственные и реальные сети*

По способу организации сети подразделяются на реальные и искусственные.

**Искусственные сети** (псевдосети) позволяют связывать компьютеры вместе через последовательные или параллельные порты и не нуждаются в дополнительных устройствах. Иногда связь в такой сети называют связью по нуль-модему (не используется модем). Само соединение называют нуль-

модемным. Искусственные сети используются когда необходимо перекачать информацию с одного компьютера на другой.

Так же эти сети используются в серверах, на которых работают виртуальные ОС для соединения их в одну сеть.

**Реальные сети** позволяют связывать компьютеры с помощью специальных устройств коммутации и физической среды передачи данных.

Основной недостаток - необходимость в дополнительных устройствах.

В дальнейшем употребляя термин компьютерная сеть будем иметь в виду реальные сети.

Все многообразие компьютерных сетей можно классифицировать по группе признаков:

1. Территориальная распространенность;
2. Тип среды передачи;
3. Топология;
4. Организация взаимодействия компьютеров.
5. Подчинение узлов

### *3.3.2 Территориальная распространенность*

По территориальной распространенности сети могут быть локальными, глобальными, и региональными.

- Локальные - это сети, перекрывающие территорию не более 10 м<sup>2</sup>
- Региональные - расположенные на территории города или области
- Глобальные на территории государства или группы государств, например, всемирная сеть Internet.

В классификации сетей существует два основных термина: LAN и WAN.

**LAN** (Local Area Network) - локальные сети, имеющие замкнутую инфраструктуру до выхода на поставщиков услуг. Термин "LAN" может описывать и маленькую офисную сеть, и сеть уровня большого завода, занимающего несколько сотен гектаров. Это может быть и как у нас на установке, спец сеть, в которой только автоматизация или VPN для офисов.

Локальные сети являются сетями закрытого типа, доступ к ним разрешен только ограниченному кругу пользователей, для которых работа в такой сети непосредственно связана с их профессиональной деятельностью.

**WAN** (Wide Area Network) - глобальная сеть, покрывающая большие географические регионы, включающие в себя как локальные сети, так и прочие телекоммуникационные сети и устройства. Пример WAN - сети с коммутацией

пакетов (Frame relay), через которую могут "разговаривать" между собой различные компьютерные сети.

Термин "корпоративная сеть" также используется в литературе для обозначения объединения нескольких сетей, каждая из которых может быть построена на различных технических, программных и информационных принципах.

Глобальные сети являются открытыми и ориентированы на обслуживание любых пользователей.

### *3.3.3 Тип среды передачи информации*

По типу среды передачи сети разделяются на: проводные: на витой паре, оптоволоконные; беспроводные с передачей информации по радиоканалам, в инфракрасном диапазоне.

### *3.3.4 Топология компьютерных сетей*

Способ соединения компьютеров в сеть называется её топологией.

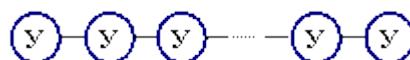
Введем определения. Узел сети представляет собой компьютер, либо коммутирующее устройство сети. **Ветвь сети** - это путь, соединяющий два смежных узла.

Узлы сети бывают трёх типов:

- оконечный узел - расположен в конце только одной ветви;
- промежуточный узел - расположен на концах более чем одной ветви;
- смежный узел - такие узлы соединены по крайней мере одним путём, не содержащим никаких других узлов.

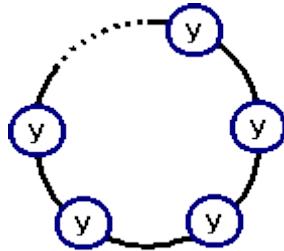
Наиболее распространенные виды топологий сетей:

#### *3.3.4.1 Линейная сеть*



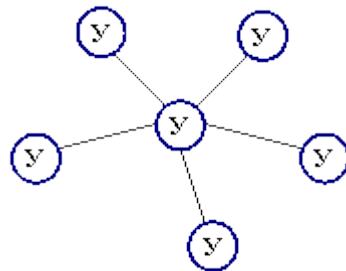
Содержит только два оконечных узла, любое число промежуточных узлов и имеет только один путь между любыми двумя узлами.

### 3.3.4.2 Кольцевая сеть



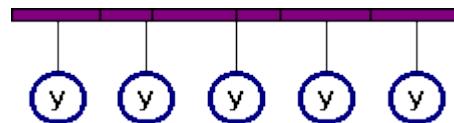
Сеть, в которой к каждому узлу присоединены две и только две ветви.

### 3.3.4.3 Звездообразная сеть



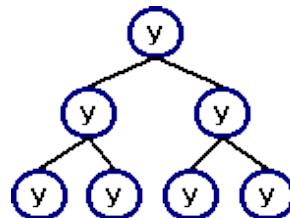
Сеть, в которой имеется только один промежуточный узел.

### 3.3.4.4 Общая шина



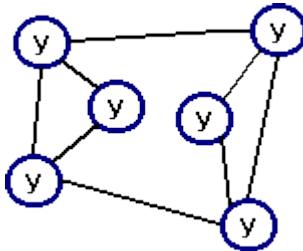
В этом случае подключение и обмен данными производится через общий канал связи, называемый общей шиной, например RS-485

### 3.3.4.5 Древовидная сеть



Сеть, которая содержит более двух оконечных узлов и по крайней мере два промежуточных узла, и в которой между двумя узлами имеется только один путь.

### *3.3.4.6 Ячеистая сеть*



Сеть, которая содержит по крайней мере два узла, имеющих два или более пути между ними.

### *3.3.4.7 Полносвязная сеть*

Сеть, в которой имеется ветвь между любыми двумя узлами.

## *3.3.5 Одноранговые и иерархические сети*

С точки зрения организации взаимодействия компьютеров, сети делят на одноранговые (Peer-to-Peer Network) и с выделенным сервером (Dedicated Server Network).

### *3.3.5.1 Одноранговые сети*

Все компьютеры одноранговой сети равноправны. Любой пользователь сети может получить доступ к данным, хранящимся на любом компьютере.

Одноранговые сети могут быть организованы с помощью таких операционных систем, как Novell Netware Lite.

Достоинства одноранговых сетей:

- Наиболее просты в установке и эксплуатации.
- Операционные системы DOS и windows обладают всеми необходимыми функциями, позволяющими строить одноранговую сеть.

Недостаток:

В условиях одноранговых сетей затруднено решение вопросов защиты информации. Поэтому такой способ организации сети используется для сетей с небольшим количеством компьютеров и там, где вопрос защиты данных не является принципиальным.

### *3.3.5.2 Иерархические сети*

В иерархической сети при установке сети заранее выделяются один или несколько компьютеров, управляющих обменом данных по сети и распределением ресурсов. Такой компьютер называют сервером.

Любой компьютер, имеющий доступ к услугам сервера называют клиентом сети или рабочей станцией.

Сервер в иерархических сетях - это постоянное хранилище разделяемых ресурсов. Сам сервер может быть клиентом только сервера более высокого уровня иерархии. Поэтому иерархические сети иногда называются сетями с выделенным сервером.

Серверы обычно представляют собой высокопроизводительные компьютеры, возможно, с несколькими параллельно работающими процессорами, с винчестерами большой емкости, с высокоскоростной сетевой картой.

Иерархическая модель сети является наиболее предпочтительной, так как позволяет создать наиболее устойчивую структуру сети и более рационально распределить ресурсы.

Также достоинством иерархической сети является более высокий уровень защиты данных.

К недостаткам иерархической сети, по сравнению с одноранговыми сетями, относятся:

- Более высокая сложность установки и модернизации сети.
- Необходимость выделения отдельного компьютера в качестве сервера.

### *3.3.5.3 Две технологии использования сервера*

Различают две технологии использования сервера:

- Технологию файл-сервера
- Архитектуру клиент-сервер.

В первой модели используется файловый сервер, на котором хранится большинство программ и данных. По требованию пользователя ему пересыпаются необходимая программа и данные. Обработка информации выполняется на рабочей станции.

В системах с архитектурой клиент-сервер обмен данными осуществляется между приложением-клиентом (front-end) и приложением-сервером (back-end). Хранение данных и их обработка производится на мощном сервере, который выполняет также контроль за доступом к ресурсам и данным. Рабочая станция получает только результаты запроса. Разработчики приложений по обработке информации обычно используют эту технологию.

Использование больших по объему и сложных приложений привело к развитию многоуровневой, в первую очередь трехуровневой архитектуры с

размещением данных на отдельном сервере базы данных (БД). Все обращения к базе данных идут через сервер приложений, где они объединяются.

### **3.4 Понятие сетевого протокола.**

Работой компьютеров в локальной сети управляют программы. Для того чтобы все компьютеры могли понимать друг друга, отправлять друг другу запросы и получать ответы, они должны общаться на одном языке. Такой язык общения компьютеров называется сетевым протоколом.

В последнее время широкое применение нашли так называемые пакетные протоколы. При использовании протоколов этого типа данные, которыми обмениваются компьютеры, режутся на небольшие блоки. Каждый блок как бы вкладывается в «конверт» (инкапсулируется), в результате чего образуется пакет. Пакет содержит как сами данные, так и служебную информацию: от кого отправлен, кому предназначен, какой пакет должен следовать за ним и прочее.

Пакетный протокол обеспечивает циркуляцию пакетов в сети, а также получение их адресатом и сборку. Каждая рабочая станция периодически подключается к сети (по прерываниям) и проверяет проходящие пакеты. Те, что адресованы ей, она забирает, а прочие пересыпает дальше.

Протокол передачи данных требует следующей информации:

- *Синхронизация* - Под синхронизацией понимают механизм распознавания начала блока данных и его конца.
- *Инициализация* - Под инициализацией понимают установление соединения между взаимодействующими партнерами.
- *Блокирование* - Под блокированием понимают разбиение передаваемой информации на блоки данных строго определенной максимальной длины (включая опознавательные знаки начала блока и его конца).
- *Адресация* - Адресация обеспечивает идентификацию различного используемого оборудования данных, которое обменивается друг с другом информацией во время взаимодействия.
- *Обнаружение ошибок* - Под обнаружением ошибок понимают установку битов четности и, следовательно, вычисление контрольных битов.
- *Нумерация блоков* - Текущая нумерация блоков позволяет установить ошибочно передаваемую или потерявшуюся информацию.

- *Управление потоком данных* - Управление потоком данных служит для распределения и синхронизации информационных потоков. Так, например, если не хватает места в буфере устройства данных или данные не достаточно быстро обрабатываются в периферийных устройствах (например, принтерах), сообщения и / или запросы накапливаются.
- *Методы восстановления* - После прерывания процесса передачи данных используют методы восстановления, чтобы вернуться к определенному положению для повторной передачи информации.
- *Разрешение доступа* - Распределение, контроль и управление ограничениями доступа к данным вменяются в обязанность пункта разрешения доступа (например, "только передача" или "только прием").

### 3.4.1 Пример

Допустим, мы на-писали письмо другу на трех листах, потом вложили их в три конверта, поставили на них цифры 1, 2, 3 и опустили в три разных почтовых ящика. Каждый листок пойдет к адресату своим путем. Возможно, что третий листок придет раньше первого, но это не помешает собрать их в правильном порядке и прочесть. В любой переписке особую роль играет конверт - необходимый элемент протокола, установленного почтовой службой. На конверте написано, куда надо доставить содержимое (адрес и почтовый индекс получателя), и указано, куда надо вернуть конверт в случае недоставки (обратный адрес). Если конверт подписан неправильно или на нем нет марки, свидетельствующей о том, что услуга оплачена, то протокол не соблюден и письмо до адресата может не дойти.

### 3.5 Семиуровневая модель OSI/ISO.

Модель OSI				
	Уровень (layer)	Тип данных (PDU <sup>[1]</sup> )	Функции	Примеры
Host layers	7. Прикладной (application)	Данные	Доступ к сетевым службам	HTTP, FTP, POP3, WebSocket
	6. Представления (presentation)		Представление и шифрование данных	ASCII, EBCDIC
	5. Сеансовый (session)		Управление сеансом связи	RPC, PAP, L2TP
	4. Транспортный (transport)	Сегменты (segment) /Дейтаграммы (datagram)	Прямая связь между конечными пунктами и надёжность	TCP, UDP, SCTP, PORTS
Media <sup>[2]</sup> layers	3. Сетевой (network)	Пакеты (packet)	Определение маршрута и логическая адресация	IPv4, IPv6, IPsec, AppleTalk
	2. Канальный (data link)	Биты (bit)/ Кадры (frame)	Физическая адресация	PPP, IEEE 802.22, Ethernet, DSL, ARP, сетевая карта.
	1. Физический (physical)	Биты (bit)	Работа со средой передачи, сигналами и двоичными данными	USB, кабель («витая пара», коаксиальный, оптоволоконный), радиоканал

Основная идея этой модели заключается в том, что каждому уровню отводится конкретная ролью в том числе и транспортной среде. Благодаря этому общая задача передачи данных расчленяется на отдельные легко обозримые задачи.

На передающей ЭВМ данные, которые необходимо передать по сети, поступают на верхний, прикладной уровень модели OSI. Далее эти данные передаются вниз по уровням модели OSI от прикладного до физического. Каждый уровень модели OSI, кроме физического, добавляет к поступившим от вышележащего уровня данным заголовок, содержащий управляющую информацию для соответствующего уровня принимающей ЭВМ. На сетевом уровне совокупная информация (данные и заголовки) делится на пакеты по числу маршрутов передачи, а на канальном уровне эти пакеты делятся на кадры стандартного размера, пригодные для непосредственной передачи на физическом уровне.

На принимающей ЭВМ поступившая по сети информация передается вверх по уровням модели OSI от физического до прикладного. Каждый уровень модели OSI, кроме физического, удаляет соответствующий адресованный ему заголовок. На канальном уровне принятые кадры упорядочиваются, так как могут быть приняты не в той последовательности, чем посылались, дополнительно запрашиваются потерянные и искаженные кадры, и из них

формируется пакет, который передается затем на вышележащий сетевой уровень. На сетевом уровне после прихода всех пакетов одного сообщения, передававшихся по разным маршрутам, они упорядочиваются, и из них формируется полная информация, переданная с другой ЭВМ. Полученные от другой ЭВМ данные, выдаются прикладным уровнем модели OSI запрашивающему процессу.

### *3.5.1 Уровень 1. Физический.*

На физическом уровне определяются электрические, механические, функциональные и процедурные параметры для физической связи в системах. Физическая связь и неразрывная с ней эксплуатационная готовность являются основной функцией 1-го уровня. Стандарты физического уровня включают рекомендации V.24 МККТТ (CCITT), EIA rS232 и X.21. Стандарт ISDN (Integrated Services Digital Network) в будущем сыграет определяющую роль для функций передачи данных. В качестве среды передачи данных используют трехжильный медный провод (экранированная витая пара), коаксиальный кабель, оптоволоконный проводник и радиорелейную линию.

### *3.5.2 Уровень 2. Канальный.*

Канальный уровень (англ. Data Link layer), также уровень передачи данных[1] — второй уровень сетевой модели OSI, предназначенный для передачи данных узлам, находящимся в том же сегменте локальной сети. Также может использоваться для обнаружения и, возможно, исправления ошибок, возникших на физическом уровне. Примерами протоколов, работающих на канальном уровне, являются: Ethernet для локальных сетей (многоузловой), Point-to-Point Protocol (PPP), HDLC и ADCCP для подключений точка-точка (двухузловой).

Канальный уровень отвечает за доставку кадров (frame) между устройствами, подключенными к одному сетевому сегменту. Кадры канального уровня не пересекают границ сетевого сегмента. Кадры передаются последовательно с обработкой кадров подтверждения, отсылаемых обратно получателем[1].

Функции межсетевой маршрутизации и глобальной адресации осуществляются на более высоких уровнях модели OSI, что позволяет протоколам канального уровня сосредоточиться на локальной доставке и адресации.

Заголовок кадра содержит аппаратные адреса MAC отправителя и получателя, что позволяет определить, какое устройство отправило кадр и какое устройство должно получить и обработать его. В отличие от иерархических и маршрутизуемых адресов, аппаратные адреса одноуровневые. Это означает, что никакая часть адреса не может указывать на принадлежность к какой-либо логической или физической группе.

Когда устройства пытаются использовать среду одновременно, возникают коллизии кадров. Протоколы канального уровня выявляют такие случаи и обеспечивают механизмы для уменьшения их количества или же их предотвращения.

Многие протоколы канального уровня не имеют подтверждения о приёме кадра, некоторые протоколы даже не имеют контрольной суммы для проверки целостности кадра. В таких случаях протоколы более высокого уровня должны обеспечивать управление потоком данных, контроль ошибок, подтверждение доставки и ретрансляции утерянных данных.

На этом уровне работают коммутаторы, мосты.

### *3.5.3 Уровень 3. Сетевой.*

Сетевой уровень (англ. Network layer) предназначается для определения пути передачи данных. Отвечает за трансляцию логических адресов и имён в физические, определение кратчайших маршрутов, коммутацию и маршрутизацию, отслеживание неполадок и заторов в сети. На этом уровне работает такое сетевое устройство, как маршрутизатор.

Максимальная длина пакета сетевого уровня может быть ограничена командой ip mtu. Рассказать про шлюз и 2 массива Rx Tx по 1500 как памяти не хватало.

### *3.5.4 Уровень 4. Транспортный.*

Транспортный уровень (англ. Transport layer) — 4-й уровень сетевой модели OSI, предназначен для доставки данных. При этом неважно, какие данные передаются, откуда и куда, то есть, он предоставляет сам механизм передачи. Блоки данных он разделяет на фрагменты, размеры которых зависят от протокола: короткие объединяет в один, а длинные разбивает. Протоколы этого уровня предназначены для взаимодействия типа точка-точка. Пример: TCP, UDP, SCTP.

Существует множество классов протоколов транспортного уровня, начиная от протоколов, предоставляющих только основные транспортные функции, например, функции передачи данных без подтверждения приема, и заканчивая протоколами, которые гарантируют доставку в пункт назначения нескольких пакетов данных в надлежащей последовательности, мультиплексируют несколько потоков данных, обеспечивают механизм управления потоками данных и гарантируют достоверность принятых данных.

Некоторые протоколы транспортного уровня, называемые протоколами без установки соединения, не гарантируют, что данные доставляются по назначению в том порядке, в котором они были посланы устройством-источником. Некоторые транспортные уровни справляются с этим, собирая данные в нужной последовательности до передачи их на сеансовый уровень. Мультиплексирование (multiplexing) данных означает, что транспортный уровень способен одновременно обрабатывать несколько потоков данных (потоки могут поступать и от различных приложений) между двумя системами. Механизм управления потоком данных — это механизм, позволяющий регулировать количество данных, передаваемых от одной системы к другой. Протоколы транспортного уровня часто имеют функцию контроля доставки данных, заставляя принимающую систему отправлять подтверждения передающей стороне о приеме данных.

### *3.5.5 Уровень 5. Сеансовый.*

Отвечает за поддержание сеанса связи, позволяя приложениям взаимодействовать между собой длительное время. Уровень управляет созданием/завершением сеанса, обменом информацией, синхронизацией задач, определением права на передачу данных и поддержанием сеанса в периоды неактивности приложений. Синхронизация передачи обеспечивается помещением в поток данных контрольных точек, начиная с которых возобновляется процесс при нарушении взаимодействия.

Сеансы передачи составляются из запросов и ответов, которые осуществляются между приложениями. Службы сеансового уровня обычно используются в средах приложений, в которых требуется использование удалённого вызова процедур.

Примером протокола сеансового уровня является NetBIOS. В случае потери или длительной соединения этот протокол может попытаться его восстановить. Если соединение не используется длительное время, то протокол

сеансового уровня может его закрыть и открыть заново. Он позволяет производить передачу в дуплексном (одновременно в обе стороны) или в полудуплексном (одновременно только в одну сторону) режимах и обеспечивает наличие контрольных точек в потоке обмена сообщениями[1].

### *3.5.6 Уровень 6. Представления данных.*

Этот уровень отвечает за преобразование протоколов и кодирование/декодирование данных. Запросы приложений, полученные с уровня приложений, он преобразует в формат для передачи по сети, а полученные из сети данные преобразует в формат, понятный приложениям. На этом важном уровне может осуществляться сжатие/распаковка или кодирование/декодирование данных, а также перенаправление запросов другому сетевому ресурсу, если они не могут быть обработаны локально.

На представительском уровне передаваемая по сети информация не меняет содержания. С помощью средств, реализованных на данном уровне, протоколы прикладных программ преодолевают синтаксические различия в представляемых данных или же различия в кодах символов, например согласовывая представления данных расширенного двоичного кода обмена информацией EBCDIC используемого мейнфреймом компании IBM с одной стороны и американского стандартного кода обмена информацией ASCII с другой.

На уровне представления реализованы следующие функции:

- преобразование данных, такое как изменение порядка битов, замена CR ("возврат каретки") на CR+LF или преобразование целого числа в число с плавающей запятой;<sup>[1]</sup>
- перевод символов, например, из кодировки ASCII в EBCDIC;<sup>[1]</sup>
- сжатие данных для увеличения пропускной способности канала;<sup>[1]</sup>
- шифрование и расшифрование.<sup>[1]</sup> Одно из применений шифрования - передача пароля принимающему компьютеру.

### *3.5.7 Уровень 7. Прикладной.*

Протокол верхнего (7-го) уровня сетевой модели OSI, обеспечивает взаимодействие сети и пользователя. Уровень разрешает приложениям пользователя иметь доступ к сетевым службам, таким, как обработчик запросов к базам данных, доступ к файлам, пересылке электронной почты. Также отвечает за передачу служебной информации, предоставляет приложениям информацию

об ошибках и формирует запросы к уровню представления. Пример: HTTP, POP3, SMTP.

### **3.6 Способы маршрутизации сообщений в сетях ЭВМ.**

<http://www.hi-edu.ru/e-books/xbook689/01/part-004.htm>

Для соединения локальных сетей используют следующие устройства, различающиеся между собой по назначению и возможностям:

- **Хаб** – пересылает пакеты от всех всем. Представляет собой простейший разветвитель, обеспечивающий прямое соединение между клиентами сети.
- **Свич** — более “умное” устройство, распределяющее пакеты данных между клиентами в соответствии с запросом. Имеет таблицу адресации, пересылает с гнезда на гнездо. (2 уровень OSI)
- **Маршрутизатор** – объединяет сети с общим протоколом более эффективнее, чем мост. Выбирают лучший путь для прохождения пакета и т.д. (3 уровень OSI)
- **Мост** - средство передачи пакетов между сетями (локальными), для протоколов сетевого уровня прозрачен. Осуществляет фильтрацию пакетов, не выпуская из сети пакеты для адресатов, находящихся внутри сети, а также переадресацию - передачу пакетов в другую сеть в соответствии с таблицей маршрутизации или во все другие сети при отсутствии адресата в таблице. Таблица маршрутизации обычно составляется в процессе самообучения по адресу источника приходящего пакета.
- **Шлюз** – программно-аппаратный комплекс, соединяющий разнородные сети или сетевые устройства. Позволяет решать проблемы различия протоколов и/или систем адресации. Действует на 5, 6 и 7-м уровнях модели OSI - сеансовом, представления и прикладном.
- **Повторитель**

### **3.7 Сетевая архитектура TCP/IP: основные принципы организации и функционирования.**

TCP/IP — сетевая модель передачи данных, представленных в цифровом виде. Модель описывает способ передачи данных от источника информации к

получателю. В модели предполагается прохождение информации через четыре уровня, каждый из которых описывается правилом (протоколом передачи). Наборы правил, решающих задачу по передаче данных, составляют стек протоколов передачи данных, на которых базируется Интернет[1][2]. Название TCP/IP происходит из двух важнейших протоколов семейства — Transmission Control Protocol (TCP) и Internet Protocol (IP), которые были первыми разработаны и описаны в данном стандарте.

TCP/IP		OSI
Прикладной	Прикладной	напр., <a href="#">HTTP</a> , <a href="#">SMTP</a> , <a href="#">SNMP</a> , <a href="#">FTP</a> , <a href="#">Telnet</a> , <a href="#">SSH</a> , <a href="#">SCP</a> , <a href="#">SMB</a> , <a href="#">NFS</a> , <a href="#">RTSP</a> , <a href="#">BGP</a>
	Представления	напр., <a href="#">XDR</a> , <a href="#">AFP</a> , <a href="#">TLS</a> , <a href="#">SSL</a>
	Сеансовый	напр., ISO 8327 / CCITT X.225, <a href="#">RPC</a> , <a href="#">NetBIOS</a> , <a href="#">PPTP</a> , <a href="#">L2TP</a> , <a href="#">ASP</a>
	Транспортный	напр., <a href="#">TCP</a> , <a href="#">UDP</a> , <a href="#">SCTP</a> , <a href="#">SPX</a> , <a href="#">ATP</a> , <a href="#">DCCP</a> , <a href="#">GRE</a>
	Сетевой	напр., <a href="#">IP</a> , <a href="#">ICMP</a> , <a href="#">IGMP</a> , <a href="#">CLNP</a> , <a href="#">OSPF</a> , <a href="#">RIP</a> , <a href="#">IPX</a> , <a href="#">DDP</a>
	Канальный	напр., <a href="#">Ethernet</a> , <a href="#">Token ring</a> , <a href="#">HDLC</a> , <a href="#">PPP</a> , <a href="#">X.25</a> , <a href="#">Frame relay</a> , <a href="#">ISDN</a> , <a href="#">ATM</a> , <a href="#">SPB</a> , <a href="#">MPLS</a> , <a href="#">ARP</a>
	Физический	напр., электрические провода, радиосвязь, волоконно-оптические провода, инфракрасное излучение

### 3.7.1 Уровни стека TCP/IP

Стек протоколов TCP/IP включает в себя четыре уровня[5]:

- Прикладной уровень (Application Layer),
- Транспортный уровень (Transport Layer),
- Межсетевой уровень (Сетевой уровень[6]) (Internet Layer),
- Канальный уровень (Network Access Layer).

Протоколы этих уровней полностью реализуют функциональные возможности модели OSI. На стеке протоколов TCP/IP построено всё взаимодействие пользователей в IP-сетях. Стек является независимым от физической среды передачи данных, благодаря чему, в частности, обеспечивается полностью прозрачное взаимодействие между проводными и беспроводными сетями.

Прикладной (Application layer)	напр., <a href="#">HTTP</a> , <a href="#">RTSP</a> , <a href="#">FTP</a> , <a href="#">DNS</a>
Транспортный (Transport Layer)	напр., <a href="#">TCP</a> , <a href="#">UDP</a> , <a href="#">SCTP</a> , <a href="#">DCCP</a> ( <a href="#">RIP</a> , протоколы маршрутизации, подобные <a href="#">OSPF</a> , что работают поверх <a href="#">IP</a> , являются частью сетевого уровня)
Сетевой (Межсетевой) (Internet Layer)	Для TCP/IP это <a href="#">IP</a> (вспомогательные протоколы, вроде <a href="#">ICMP</a> и <a href="#">IGMP</a> , работают поверх <a href="#">IP</a> , но тоже относятся к сетевому уровню; протокол <a href="#">ARP</a> является самостоятельным вспомогательным протоколом, работающим поверх канального уровня)
Уровень сетевого доступа (Канальный) (Network Access Layer)	Ethernet, IEEE 802.11 WLAN, SLIP, Token Ring, ATM и <a href="#">MPLS</a> , физическая среда и принципы кодирования информации, T1, E1

### *3.7.2 Прикладной уровень*

На прикладном уровне (Application layer) работает большинство сетевых приложений.

Эти программы имеют свои собственные протоколы обмена информацией, например, интернет браузер для протокола HTTP, ftp-клиент для протокола FTP (передача файлов), почтовая программа для протокола SMTP (электронная почта), SSH (безопасное соединение с удалённой машиной), DNS (преобразование символьных имён в IP-адреса) и многие другие.

В массе своей эти протоколы работают поверх TCP или UDP и привязаны к определённому порту, например:

- HTTP на TCP-порт 80 или 8080,
- FTP на TCP-порт 20 (для передачи данных) и 21 (для управляющих команд),
- SSH на TCP-порт 22,
- запросы DNS на порт UDP (реже TCP) 53,
- обновление маршрутов по протоколу RIP на UDP-порт 520.

Эти порты определены Агентством по выделению имен и уникальных параметров протоколов (IANA).

К этому уровню относятся: HTTP, NTP (время), POP3 (почта), SSH.

### *3.7.3 Транспортный уровень*

Протоколы транспортного уровня (Transport layer) могут решать проблему негарантированной доставки сообщений («дошло ли сообщение до адресата?»), а также гарантировать правильную последовательность прихода данных. В стеке TCP/IP транспортные протоколы определяют, для какого именно приложения предназначены эти данные.

TCP (IP идентификатор 6) — «гарантированный» транспортный механизм с предварительным установлением соединения, предоставляющий приложению надёжный поток данных, дающий уверенность в безошибочности получаемых данных, перезапрашивающий данные в случае потери и устраниющий дублирование данных. TCP позволяет регулировать нагрузку на сеть, а также уменьшать время ожидания данных при передаче на большие расстояния. Более того, TCP гарантирует, что полученные данные были отправлены точно в такой же последовательности. В этом его главное отличие от UDP.

UDP (IP идентификатор 17) протокол передачи датаграмм без установления соединения. Также его называют протоколом «ненадёжной»

передачи, в смысле невозможности удостовериться в доставке сообщения адресату, а также возможного перемешивания пакетов. В приложениях, требующих гарантированной передачи данных, используется протокол TCP.

UDP обычно используется в таких приложениях, как потоковое видео и компьютерные игры, где допускается потеря пакетов, а повторный запрос затруднён или не оправдан, либо в приложениях вида запрос-ответ (например, запросы к DNS), где создание соединения занимает больше ресурсов, чем повторная отправка.

И TCP, и UDP используют для определения протокола верхнего уровня число, называемое портом.

### *3.7.4 Сетевой (межсетевой) уровень*

Межсетевой уровень (Internet layer) изначально разработан для передачи данных из одной сети в другую. На этом уровне работают маршрутизаторы, которые перенаправляют пакеты в нужную сеть путём расчёта адреса сети по маске сети. Примерами такого протокола является X.25 и IPC в сети ARPANET.

С развитием концепции глобальной сети в уровень были внесены дополнительные возможности по передаче из любой сети в любую сеть, независимо от протоколов нижнего уровня, а также возможность запрашивать данные от удалённой стороны, например в протоколе ICMP (используется для передачи диагностической информации IP-соединения) и IGMP (используется для управления multicast-потоками).

ICMP и IGMP расположены над IP и должны попасть на следующий — транспортный — уровень, но функционально являются протоколами сетевого уровня, и поэтому их невозможно вписать в модель OSI.

Пакеты сетевого протокола IP могут содержать код, указывающий, какой именно протокол следующего уровня нужно использовать, чтобы извлечь данные из пакета. Это число — уникальный IP-номер протокола. ICMP и IGMP имеют номера, соответственно, 1 и 2.

**ICMP** (англ. Internet Control Message Protocol — протокол межсетевых управляющих сообщений[1]) — сетевой протокол, входящий в стек протоколов TCP/IP. В основном ICMP используется для передачи сообщений об ошибках и других исключительных ситуациях, возникших при передаче данных, например, запрашиваемая услуга недоступна, или хост, или маршрутизатор не отвечают. Также на ICMP возлагаются некоторые сервисные функции.

**IGMP** (англ. Internet Group Management Protocol — протокол управления группами Интернета) — протокол управления групповой (multicast) передачей данных в сетях, основанных на протоколе IP. IGMP используется маршрутизаторами и IP-узлами для организации сетевых устройств в группы.

Этот протокол является частью спецификации групповой передачи пакетов в IP-сетях. IGMP расположен на сетевом уровне[1]. Он во многом аналогичен ICMP для одноадресной передачи. IGMP может использоваться для поддержки потокового видео и онлайн-игр, для этих типов приложений он позволяет использовать сетевые ресурсы более эффективно. IGMP уязвим для некоторых атак[2][3][4][5], и брандмауэры обычно позволяют пользователю отключить этот протокол, если в нем нет необходимости.

К этому уровню относятся: DVMRP, ICMP, IGMP, MARS, PIM, RIP, RIP2, RSVP

### 3.7.5 Канальный уровень

Канальный уровень (Link layer) описывает способ кодирования данных для передачи пакета данных на физическом уровне (то есть специальные последовательности бит, определяющих начало и конец пакета данных, а также обеспечивающие помехоустойчивость). Ethernet, например, в полях заголовка пакета содержит указание того, какой машине или машинам в сети предназначен этот пакет.

Примеры протоколов канального уровня — Ethernet, IEEE 802.11 WLAN, SLIP, Token Ring, ATM и MPLS.

Кроме того, канальный уровень описывает среду передачи данных (будь то коаксиальный кабель, витая пара, оптическое волокно или радиоканал), физические характеристики такой среды и принцип передачи данных (разделение каналов, модуляцию, амплитуду сигналов, частоту сигналов, способ синхронизации передачи, время ожидания ответа и максимальное расстояние).

При проектировании стека протоколов на канальном уровне рассматривают помехоустойчивое кодирование — позволяющие обнаруживать и исправлять ошибки в данных вследствие воздействия шумов и помех на канал связи.

Многие операционные системы успешно работают на различных аппаратных платформах без существенных изменений в своем составе. Во многом это объясняется тем, что, несмотря на различия в деталях, средства аппаратной поддержки ОС большинства компьютеров приобрели сегодня много типовых черт, а именно эти средства в первую очередь влияют на работу компонентов операционной системы. В результате в ОС можно выделить достаточно компактный слой машинно-зависимых компонентов ядра и сделать остальные слои ОС общими для разных аппаратных платформ.

Четкой границы между программной и аппаратной реализацией функций ОС не существует — решение о том, какие функции ОС будут выполняться программно, а какие аппаратно, принимается разработчиками аппаратного и программного обеспечения компьютера. Тем не менее практически все современные аппаратные платформы имеют некоторый типичный набор средств аппаратной поддержки ОС, в который входят следующие компоненты:

## 4.1 Система прерываний

Прерывание - это прекращение выполнения текущей команды или текущей последовательности команд для обработки некоторого события специальной программой - обработчиком прерывания, с последующим возвратом к выполнению прерванной программы. Событие может быть вызвано особой ситуацией, сложившейся при выполнении программы, или сигналом от внешнего устройства. Прерывание используется для быстрой реакции процессора на особые ситуации, возникающие при выполнении программы и взаимодействии с внешними устройствами.

Механизм прерывания обеспечивается соответствующими аппаратно-программными средствами компьютера.

Любая особая ситуация, вызывающая прерывание, сопровождается сигналом, называемым запросом прерывания (ЗП). Запросы прерываний от внешних устройств поступают в процессор по специальным линиям, а запросы, возникающие в процессе выполнения программы, поступают непосредственно изнутри микропроцессора. Механизмы обработки прерываний обоих типов схожи. Рассмотрим функционирование компьютера при появлении сигнала запроса прерывания, опираясь в основном на обработку аппаратных прерываний.

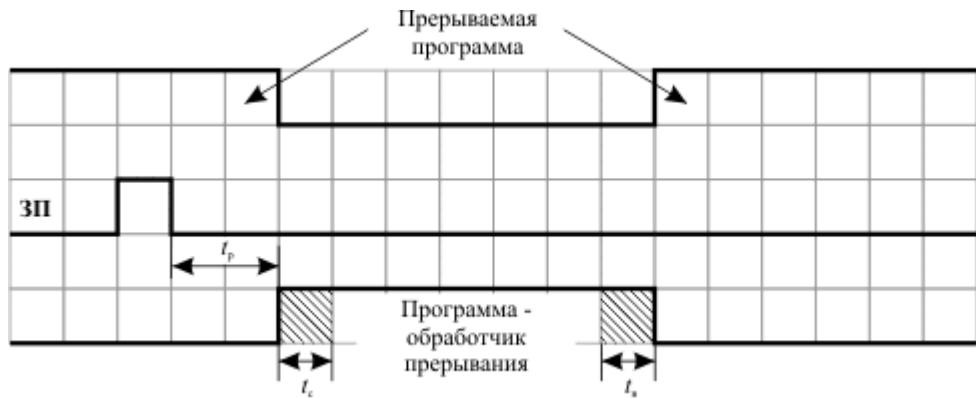


Рисунок 5 Выполнение прерывания в компьютере:  $t_p$  - время реакции процессора на запрос прерывания;  $t_c$  - время сохранения состояния прерываемой программы и вызова обработчика прерывания;  $t_b$  - время восстановления прерванной программы

#### 4.1.1 Время реакции

Время реакции - это время между появлением сигнала запроса прерывания и началом выполнения прерывающей программы (обработчика прерывания) в том случае, если данное прерывание разрешено к обслуживанию.

Время реакции зависит от момента, когда процессор определяет факт наличия запроса прерывания. Опрос запросов прерываний может проводиться либо по окончании выполнения очередного этапа команды (например, считывание команды, считывание первого операнда и т.д.), либо после завершения каждой команды программы.

Первый подход обеспечивает более быструю реакцию, но при этом необходимо при переходе к обработчику прерывания сохранять большой объем информации о прерываемой программе, включающей состояние буферных регистров процессора, номера завершившегося этапа и т.д. При возврате из обработчика также необходимо выполнить большой объем работы по восстановлению состояния процессора.

Во втором случае время реакции может быть достаточно большим. Однако при переходе к обработчику прерывания требуется запоминание минимального контекста прерываемой программы (обычно это счетчик команд и регистр флагов). В настоящее время в компьютерах чаще используется распознавание запроса прерывания после завершения очередной команды.

Время реакции определяется для запроса с наивысшим приоритетом.

#### 4.1.2 Глубина прерывания

Глубина прерывания - максимальное число программ, которые могут прерывать друг друга. Глубина прерывания обычно совпадает с числом уровней приоритетов, распознаваемых системой прерываний. Работа системы прерываний при различной глубине прерываний ( $n$ ) представлена на рис. 14.2. Здесь предполагается, что с увеличением номера запроса прерывания увеличивается его приоритет.

Запросы прерываний	1	2	3					
Прерываемая программа	Обработчик 1 (t1)	Обработчик 2 (t2)	Обработчик 3 (t3)	Прерываемая программа				

a)  $n = 1$

$$t1_1 + t1_2 = t1,$$

			t3					
		t2 <sub>1</sub>			t2 <sub>2</sub>			
	t1 <sub>1</sub>					t1 <sub>2</sub>		
Прерываемая программа						Прерываемая программа		

б)  $n = 3$

$$t2_1 + t2_2 = t2.$$

Рисунок 6 Работа системы прерываний при различной глубине прерываний.

#### 4.1.3 Типы прерываний

##### 4.1.3.1 Аппаратные

Аппаратные прерывания используются для организации взаимодействия с внешними устройствами. Запросы аппаратных прерываний поступают на специальные входы микропроцессора. Они бывают:

- маскируемые, которые могут быть замаскированы программными средствами компьютера. Запрос от которого можно программным образом замаскировать путем сброса флага IF в регистре флагов
- немаскируемые, запрос от которых таким образом замаскирован быть не может. Используется обычно для запросов прерываний по нарушению питания;

##### 4.1.3.2 Программные

Программные прерывания вызываются следующими ситуациями:

- особый случай, возникший при выполнении команды и препятствующий нормальному продолжению программы

(переполнение, нарушение защиты памяти, отсутствие нужной страницы в оперативной памяти и т.п.);

- наличие в программе специальной команды прерывания INT n (interrupt), используемой обычно программистом при обращениях к специальным функциям операционной системы для ввода-вывода информации.

Каждому запросу прерывания в компьютере присваивается свой номер (тип прерывания), используемый для определения адреса обработчика прерывания.

При поступлении запроса прерывания компьютер выполняет следующую последовательность действий:

- определение наиболее приоритетного незамаскированного запроса на прерывание (если одновременно поступило несколько запросов);
- определение типа выбранного запроса ;
- сохранение текущего состояния счетчика команд и регистра флагов;
- определение адреса обработчика прерывания по типу прерывания и передача управления первой команде этого обработчика;
- выполнение программы - обработчика прерывания ;
- восстановление сохранных значений счетчика команд и регистра флагов прерванной программы;
- продолжение выполнения прерванной программы.

Этапы 1-4 выполняются аппаратными средствами ЭВМ автоматически при появлении запроса прерывания. Этап 6 также выполняется аппаратно по команде возврата из обработчика прерывания.

Задача программиста - составить программу - обработчик прерывания, которая выполняла бы действия, связанные с появлением запроса данного типа, и поместить адрес начала этой программы в специальной таблице адресов прерываний. Программа-обработчик, как правило, должна начинаться с сохранения состояния тех регистров процессора, которые будут ею изменяться, и заканчиваться восстановлением состояния этих регистров. Программа-обработчик должна завершаться специальной командой, указывающей процессору на необходимость возврата в прерванную программу.

Распознавание наличия сигналов запроса прерывания и определение наиболее приоритетного из них может проводиться различными методами. Рассмотрим один из них.

#### 4.1.4 Обработчик прерывания

Переход к соответствующему обработчику прерывания осуществляется (в реальном режиме работы микропроцессора) посредством таблицы векторов прерываний. Эта таблица располагается в самых младших адресах оперативной памяти, имеет объем 1 Кбайт и содержит значения сегментного регистра команд (CS) и указателя команд (IP) для 256 обработчиков прерываний.

Обращение к элементам таблицы осуществляется по 8-разрядному коду - типу прерывания.

Тип прерывания	Источник прерывания	Vector No.	Program Address	Source	Interrupt Definition
0	Деление на 0	1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset
1	Пошаговый режим выполнения программы	2	0x0001	INT0	External Interrupt Request 0
2	Запрос по входу <code>NMI</code>	3	0x0002	PCINT0	Pin Change Interrupt Request 0
xxx		4	0x0003	TIMO_OVF	Timer/Counter Overflow
8	Запрос по входу <code>IRQ0</code> (системный таймер)	5	0x0004	EE_RDY	EEPROM Ready
9	Запрос по входу <code>IRQ1</code> (контроллер клавиатуры)	6	0x0005	ANA_COMP	Analog Comparator
xxx		7	0x0006	TIMO_COMPA	Timer/Counter Compare Match A
11	Отсутствие сегмента в оперативной памяти	8	0x0007	TIMO_COMPB	Timer/Counter Compare Match B
xxx		9	0x0008	WDT	Watchdog Time-out
255	Пользовательское прерывание	10	0x0009	ADC	ADC Conversion Complete

## 4.2 Защита памяти

При мультипрограммном режиме работы ЭВМ в ее памяти одновременно могут находиться несколько независимых программ. Поэтому необходимы специальные меры по предотвращению или ограничению обращений одной программы к областям памяти, используемым другими программами. Программы могут также содержать ошибки, которые, если этому не воспрепятствовать, приводят к искажению информации, принадлежащей другим программам. Последствия таких ошибок особенно опасны, если разрушению подвергнутся программы операционной системы. Другими словами, надо исключить воздействие программы пользователя на работу программ других пользователей и программ операционной системы. Следует защищать и сами программы от находящихся в них возможных ошибок.

Таким образом, средства защиты памяти должны предотвращать

- неразрешенное взаимодействие пользователей друг с другом,
- несанкционированный доступ пользователей к данным,
- повреждение программ и данных из-за ошибок в программах,

- намеренные попытки разрушить целостность системы,
- использование информации в памяти не в соответствии с ее функциональным назначением.

Чтобы воспрепятствовать разрушению одних программ другими, достаточно защитить область памяти данной программы от попыток записи в нее со стороны других программ, а в некоторых случаях и своей программы (защита от записи), при этом допускается обращение других программ к этой области памяти для считывания данных.

В других случаях, например, при ограничениях на доступ к информации, хранящейся в системе, необходимо запрещать другим программам любое обращение к некоторой области памяти как на запись, так и на считывание. Такая защита от записи и считывания помогает в отладке программы, при этом осуществляется контроль каждого случая обращения за область памяти своей программы.

Если нарушается защита памяти, исполнение программы приостанавливается и вырабатывается запрос прерывания по нарушению защиты памяти.

Задача от вторжения программ в чужие области памяти может быть организована различными методами. Но при любом подходе реализация защиты не должна заметно снижать производительность компьютера и требовать слишком больших аппаратурных затрат.

Методы защиты базируются на некоторых классических подходах, которые получили свое развитие в архитектуре современных ЭВМ. К таким методам можно отнести защиту отдельных ячеек, метод граничных регистров, метод ключей защиты.

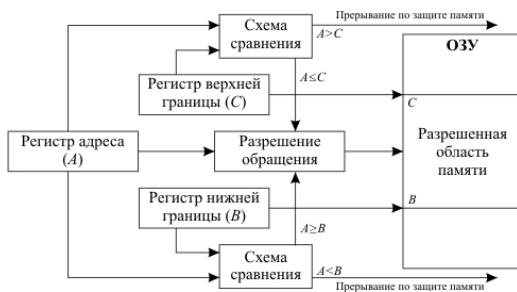
#### *4.2.1 Защита отдельных ячеек*

Защита отдельных ячеек памяти организуется в ЭВМ, предназначенных для работы в системах управления, где необходимо обеспечить возможность отладки новых программ без нарушения функционирования находящихся в памяти рабочих программ, управляющих технологическим процессом. Это может быть достигнуто выделением в каждой ячейке памяти специального "разряда защиты". Установка этого разряда в "1" запрещает производить запись в данную ячейку, что обеспечивает сохранение рабочих программ. Недостаток такого подхода - большая избыточность в кодировании информации из-за излишне мелкого уровня защищаемого объекта (ячейка).

В системах с мультипрограммной обработкой целесообразно организовывать защиту на уровне блоков памяти, выделяемых программам, а не отдельных ячеек.

#### 4.2.2 Метод граничных регистров

Метод граничных регистров (рис. 17.1) заключается во введении двух граничных регистров, указывающих верхнюю и нижнюю границы области памяти, куда программа имеет право доступа.



При каждом обращении к памяти проверяется, находится ли используемый адрес в установленных границах. При выходе за границы обращение к памяти не производится, а формируется запрос прерывания, передающий управление операционной системе. Содержание граничных регистров устанавливается операционной системой при загрузке программы в память.

Модификация этого метода заключается в том, что один регистр используется для указания адреса начала защищаемой области, а другой содержит длину этой области.

Метод граничных регистров, обладая несомненной простотой реализации, имеет и определенные недостатки. Основным из них является то, что этот метод поддерживает работу лишь с непрерывными областями памяти.

#### 4.2.3 Метод ключей защиты

Метод ключей защиты, в отличие от предыдущего, позволяет реализовать доступ программы к областям памяти, организованным в виде отдельных модулей, не представляющих собой единый массив.

Память в логическом отношении делится на одинаковые блоки, например, страницы. Каждому блоку памяти ставится в соответствие код, называемый ключом защиты памяти, а каждой программе, принимающей участие в мультипрограммной обработке, присваивается код ключа программы. Доступ программы к данному блоку памяти для чтения и записи разрешен, если ключи совпадают (то есть данный блок памяти относится к данной программе) или один

из них имеет код 0 (код 0 присваивается программам операционной системы и блокам памяти, к которым имеют доступ все программы: общие данные, совместно используемые подпрограммы и т. п.). Коды ключей защиты блоков памяти и ключей программ устанавливаются операционной системой.

В ключе защиты памяти предусматривается дополнительный разряд режима защиты. Если 0 – запрещена только запись. Если 1 – запрещено чтение и запись. Коды ключей защиты памяти хранятся в специальной памяти ключей защиты, более быстродействующей, чем оперативная память.

#### *4.2.4 Защита по привилегиям*

Защита по привилегиям фиксирует более тонкие ошибки, связанные, в основном, с разграничением прав доступа к той или иной информации.

В какой-то степени защиту по привилегиям можно сравнить с классическим методом ключей защиты памяти. Различным объектам (программам, сегментам памяти, запросам на обращение к памяти и к внешним устройствам), которые должны быть распознаны процессором, присваивается идентификатор, называемый уровнем привилегий. Процессор постоянно контролирует, имеет ли текущая программа достаточные привилегии, чтобы

- выполнять некоторые команды,
- выполнять команды ввода-вывода на том или ином внешнем устройстве,
- обращаться к данным других программ,
- вызывать другие программы.

На аппаратном уровне в процессоре различаются 4 уровня привилегий. Наиболее привилегированными являются программы на уровне 0.

Число программ, которые могут выполняться на более высоком уровне привилегий, уменьшается от уровня 3 к уровню 0. Программы уровня 0 действуют как ядро операционной системы.

- уровень 0 - ядро ОС, обеспечивающее инициализацию работы, управление доступом к памяти, защиту и ряд других жизненно важных функций, нарушение которых полностью выводит из строя процессор;
- уровень 1 - основная часть программ ОС (утилиты);
- уровень 2 - служебные программы ОС (драйверы, СУБД, специализированные подсистемы программирования и др.);
- уровень 3 - прикладные программы пользователя.

Конкретная операционная система не обязательно должна поддерживать все четыре уровня привилегий. Так, ОС UNIX работает с двумя кольцами защиты: супервизор (уровень 0) и пользователь (уровни 1,2,3).

### **4.3 Механизм преобразования адресов в системах виртуальной памяти**

Все доступное множество адресов элементов хранения, упорядоченное по какому-либо признаку, называют адресным пространством памяти. Физическое адресное пространство организовано просто как одномерный массив ячеек, каждой из которых присвоен свой номер, называемый физическим адресом.

В общем случае, под адресом понимают некий идентификатор, однозначно определяющий расположение элемента хранения среди прочих элементов в составе среды хранения.

Для адресации данных в физическом адресном пространстве программы используют логическую адресацию. Процессор автоматически транслирует логические адреса в физические, выдаваемые на адресную шину и воспринимаемые схемами управления (контроллерами) памяти.

Существуют две стратегии распределения оперативной памяти, как и любого ресурса:

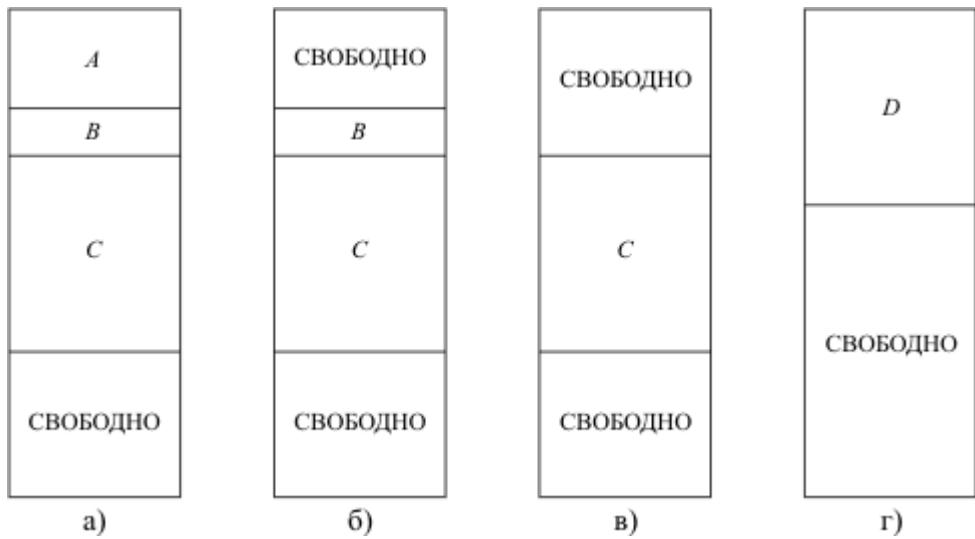
- статическое
- динамическое.

#### *4.3.1 Статическое распределение*

При статическом распределении вся необходимая оперативная память выделяется процессу в момент его создания. При этом память выделяется единым блоком необходимой длины, начало которого определяется базовым адресом. Программа пишется в адресах относительно начала блока, а физический адрес команды или операнда при выполнении программы формируется как сумма базового адреса блока и относительного адреса в блоке. Значение базового адреса устанавливается при загрузке программы в оперативную память. Так как в разных программах используются блоки разной длины, то при таком подходе возникает проблема фрагментации памяти, то есть возникают свободные участки памяти, которые невозможно без предварительного преобразования использовать для вычислительного процесса.

Пусть ОП имеет объем 10 Мбайт, а для выполнения программ А, В, С, D требуются следующие объемы памяти: А - 2 Мбайт, В - 1 Мбайт, С - 4 Мбайт, D

- 4 Мбайт. Начальное распределение памяти и распределение памяти после выполнения некоторых программ представлено на рис. ниже



Из рисунка видно, что программа D при статическом распределении памяти может быть загружена в оперативную память лишь после завершения выполнения всех предыдущих программ, хотя необходимый для нее объем памяти существовал уже после завершения работы программы А. В то же время для улучшения показателей работы мультипрограммной ЭВМ требуется, чтобы в оперативной памяти постоянно находилось возможно большее количество программ, готовых к выполнению.

Данную проблему можно частично решить перераспределением памяти после завершения выполнения каждой программы с целью формирования единого свободного участка, который может быть выделен новой программе, поступающей на обработку (дефрагментация памяти). Однако это требует трудоемкой работы системных средств и практически не используется.

#### 4.3.2 Динамическое распределение

При динамическом распределении памяти каждой программе в начальный момент выделяется лишь часть от всей необходимой ей памяти, а остальная часть выделяется по мере возникновения реальной потребности в ней. Такой подход базируется на следующих предпосылках.

**Во-первых**, при каждом конкретном исполнении в зависимости от исходных данных некоторые части программы (до 25% ее длины) вообще не используются. Следует стремиться к тому, чтобы эти фрагменты кода не загружались в ОП.

**Во-вторых**, исполнение программы характеризуется так называемым принципом локальности ссылок. Он подразумевает, что при исполнении

программы в течение некоторого относительно малого интервала времени происходит обращение к памяти в пределах ограниченного диапазона адресов (как по коду программы, так и по данным). Следовательно, на протяжении этого времени нет необходимости хранить в ОП другие блоки программы.

При этом системные средства должны отслеживать возникновение требований на обращение к тем частям программы, которые в данный момент отсутствуют в ОЗУ, выделять этой программе необходимый блок памяти и помещать туда из внешнего ЗУ требуемую часть программы. Для этого может потребоваться предварительное перемещение некоторых блоков информации из ОЗУ во внешнюю память. Данные перемещения должны быть скрыты от пользователя и в наименьшей степени замедлять работу его программы.

Динамическое распределение памяти тесно переплетается с понятием виртуальной памяти.

Принцип виртуальной памяти предполагает, что пользователь при подготовке своей программы имеет дело не с физической ОП, действительно работающей в составе компьютера и имеющей некоторую фиксированную емкость, а с виртуальной (кажущейся) одноуровневой памятью, емкость которой равна всему адресному пространству

#### *4.3.3 Страницы*

Для преобразования виртуальных адресов в физические физическая и виртуальная память разбиваются на блоки фиксированной длины, называемые страницами. Объемы виртуальной и физической страниц совпадают. Страницы виртуальной и физической памяти нумеруются.

Виртуальный (логический) адрес в этом случае представляет собой номер виртуальной страницы и смещение внутри этой страницы. В свою очередь, физический адрес - это номер физической страницы и смещение в ней.

Вначале в ОП загружается первая страница программы и ей передается управление. Когда в ходе выполнения программы происходит обращение за пределы загруженной страницы, операционная система прерывает выполнение данной программы, загружает требуемую страницу в ОП, после чего передает управление прерванной программе.

Правила перевода номеров виртуальных страниц в номера физических страниц обычно задаются в виде таблицы страничного преобразования. Такие таблицы формируются системой управления памятью и модифицируются каждый раз при перераспределении памяти.

## **4.4 Управление периферийными устройствами.**

Любая ЭВМ представляет собой сложную систему, включающую в себя большое количество различных устройств. Связь устройств ЭВМ между собой осуществляется с помощью сопряжений, которые в вычислительной технике называются интерфейсами.

Интерфейс - это совокупность программных и аппаратных средств, предназначенных для передачи информации между компонентами ЭВМ и включающих в себя электронные схемы, линии, шины и сигналы адресов, данных и управления, алгоритмы передачи сигналов и правила интерпретации сигналов устройствами.

Интерфейсы характеризуются следующими параметрами:

- пропускная способность - количество информации, которая может быть передана через интерфейс в единицу времени;
- максимальная частота передачи информационных сигналов через интерфейс ;
- максимально допустимое расстояние между соединяемыми устройствами;
- общее число проводов (линий) в интерфейсе ;
- информационная ширина интерфейса - число бит или байт данных, передаваемых параллельно через интерфейс.

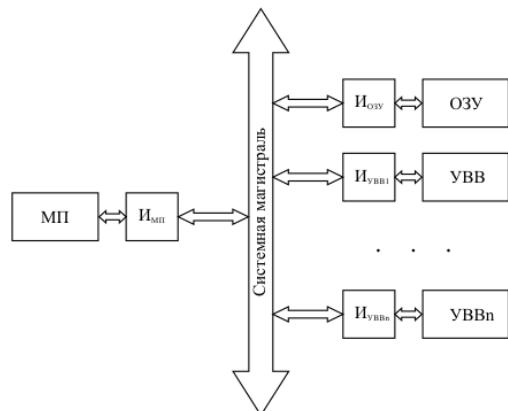
К динамическим параметрам интерфейса относится время передачи отдельного слова и блока данных с учетом продолжительности процедур подготовки и завершения передачи.

Разработка систем ввода-вывода требует решения целого ряда проблем, среди которых выделим следующие:

- необходимо обеспечить возможность реализации ЭВМ с переменным составом оборудования, в первую очередь, с различным набором устройств ввода-вывода, с тем, чтобы пользователь мог выбирать конфигурацию машины в соответствии с ее назначением, легко добавлять новые устройства и отключать те, в использовании которых он не нуждается;
- для эффективного и высокопроизводительного использования оборудования компьютера следует реализовать параллельную во времени работу процессора над вычислительной частью программы и выполнение периферийными устройствами процедур ввода-вывода;

- необходимо упростить для пользователя и стандартизовать программирование операций ввода-вывода, обеспечить независимость программирования ввода-вывода от особенностей того или иного периферийного устройства;
- в ЭВМ должно быть обеспечено автоматическое распознавание и реакция процессора на многообразие ситуаций, возникающих в УВВ (готовность устройства, отсутствие носителя, различные нарушения нормальной работы и др.).

Главным направлением решения указанных проблем является магистрально-модульный способ построения ЭВМ рис.: все устройства, составляющие компьютер, включая и микропроцессор, организуются в виде модулей, которые соединяются между собой общей магистралью. Обмен информацией по магистрали удовлетворяет требованиям некоторого общего интерфейса, установленного для магистрали данного типа. Каждый модуль подключается к магистрали посредством специальных интерфейсных схем:



На интерфейсные схемы модулей возлагаются следующие задачи:

- обеспечение функциональной и электрической совместимости сигналов и протоколов обмена модуля и системной магистрали;
- преобразование внутреннего формата данных модуля в формат данных системной магистрали и обратно;
- обеспечение восприятия единых команд обмена информацией и преобразование их в последовательность внутренних управляющих сигналов.

Эти интерфейсные схемы могут быть достаточно сложными и по своим возможностям соответствовать универсальным микропроцессорам. Такие схемы принято называть контроллерами.

Контроллеры обладают высокой степенью автономности, что позволяет обеспечить параллельную во времени работу периферийных устройств и выполнение программы обработки данных микропроцессором.

Недостатком магистрально-модульного способа организации ЭВМ является невозможность одновременного взаимодействия более двух модулей, что ставит ограничение на производительность компьютера. Поэтому этот способ, в основном, используется в ЭВМ, к характеристикам которых не предъявляется очень высоких требований, например в персональных ЭВМ.

В ЭВМ используются два основных способа организации передачи данных между памятью и периферийными устройствами: программно-управляемая передача и прямой доступ к памяти (ПДП).

#### *4.4.1 Программно-управляемая передача*

Программно-управляемая передача данных осуществляется при непосредственном участии и под управлением процессора. Например, при пересылке блока данных из периферийного устройства в оперативную память процессор должен выполнить следующую последовательность шагов:

1. сформировать начальный адрес области обмена ОП;
2. занести длину передаваемого массива данных в один из внутренних регистров, который будет играть роль счетчика;
3. выдать команду чтения информации из УВВ (устройство ввода-вывода); при этом на шину адреса из МП выдается адрес УВВ, на шину управления - сигнал чтения данных из УВВ, а считанные данные заносятся во внутренний регистр МП;
4. выдать команду записи информации в ОП; при этом на шину адреса из МП выдается адрес ячейки оперативной памяти, на шину управления - сигнал записи данных в ОП, а на шину данных выставляются данные из регистра МП, в который они были помещены при чтении из УВВ;
5. модифицировать регистр, содержащий адрес оперативной памяти;
6. уменьшить счетчик длины массива на длину переданных данных;
7. если переданы не все данные, то повторить шаги 3-6, в противном случае закончить обмен.

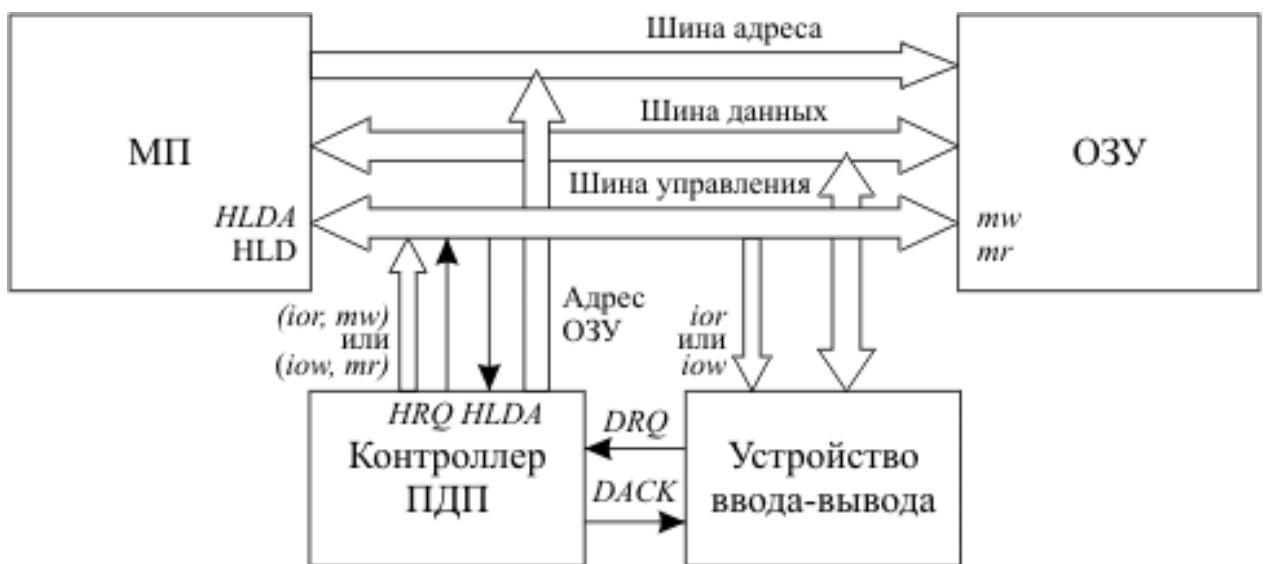
Как видно, программно-управляемый обмен ведет к нерациональному использованию мощности микропроцессора, который вынужден выполнять большое количество относительно простых операций, приостанавливая работу над основной программой. При этом действия, связанные с обращением к

оперативной памяти и к периферийному устройству, обычно требуют удлиненного цикла работы микропроцессора из-за их более медленной по сравнению с микропроцессором работы, что приводит к еще более существенным потерям производительности ЭВМ.

#### 4.4.2 Прямой доступ к памяти (DMA)

Альтернативой программно-управляемому обмену служит прямой доступ к памяти - способ быстродействующего подключения внешнего устройства, при котором оно обращается к оперативной памяти, не прерывая работы процессора [ 3 ]. Такой обмен происходит под управлением отдельного устройства - контроллера прямого доступа к памяти (КПДП).

Чем Stm32 лучше avr/Arduno.



Перед началом работы контроллер ПДП необходимо инициализировать: занести начальный адрес области ОП, с которой производится обмен, и длину передаваемого массива данных. В дальнейшем по сигналу запроса прямого доступа контроллер фактически выполняет все те действия, которые обеспечивал микропроцессор при программно-управляемой передаче.

Последовательность действий КПДП при запросе на прямой доступ к памяти со стороны устройства ввода-вывода следующая:

1. Принять запрос на ПДП (прямой доступ к памяти) (сигнал DRQ) от УВВ.
2. Сформировать запрос к МП на захват шин (сигнал HRQ).
3. Принять сигнал от МП (HLDA), подтверждающий факт перевода микропроцессором своих шин в третье состояние.

4. Сформировать сигнал, сообщающий устройству ввода-вывода о начале выполнения циклов прямого доступа к памяти (DACK).
5. Сформировать на шине адреса компьютера адрес ячейки памяти, предназначеннной для обмена.
6. Выработать сигналы, обеспечивающие управление обменом (IOR, MW для передачи данных из УВВ в оперативную память и IOW, MR для передачи данных из оперативной памяти в УВВ).
7. Уменьшить значение в счетчике данных на длину переданных данных.
8. Проверить условие окончания сеанса прямого доступа (обнуление счетчика данных или снятие сигнала запроса на ПДП). Если условие окончания не выполнено, то изменить адрес в регистре текущего адреса на длину переданных данных и повторить шаги 5-8.

Прямой доступ к памяти позволяет осуществлять параллельно во времени выполнение процессором программы и обмен данными между периферийным устройством и оперативной памятью.

Обычно программно-управляемый обмен используется в ЭВМ для операций ввода-вывода отдельных байт (слов), которые выполняются быстрее, чем при ПДП, так как исключаются потери времени на инициализацию контроллера ПДП, а в качестве основного способа осуществления операций ввода-вывода используют ПДП. Например, в стандартной конфигурации персональной ЭВМ обмен между накопителями на магнитных дисках и оперативной памятью происходит в режиме прямого доступа.

Как отмечалось выше, обычно компьютер строится по магистрально-модульному принципу. При этом все составляющие его устройства объединяются общей шиной, по которой между ними происходит обмен данными, адресами, а также управляющими сигналами. В качестве примера перечислим основные линии, составляющие одну из распространенных системных магистралей - шину ISA:

- A0-A23 - шина адреса;
- D0-D15 - двунаправленная шина данных, допускает обмен как байтами, так и словами (2 байта);
- CLK - шинный тактовый сигнал, синхронизирует работу процессора, ОП и УВВ;
- MR - управляющий сигнал чтения из ОП;
- MW - управляющий сигнал записи в ОП;
- IOR - управляющий сигнал чтения из УВВ;

- IOW - управляющий сигнал записи в УВВ;
- IRQ<sub>i</sub> - запрос прерывания от i -го источника;
- DRQ<sub>i</sub> - запрос прямого доступа к памяти по i -му каналу контроллера ПДП ;
- DACK<sub>i</sub> - разрешение прямого доступа к памяти i -му каналу контроллера ПДП ;
- AEN - сигнал занятости шин обменом в режиме ПДП,
- READY - сигнал готовности УВВ к обмену.

Магистраль обеспечивает подключение до семи внешних устройств, работающих в режиме прямого доступа к памяти, и до 11 запросов прерываний от УВВ. Еще четыре запроса прерываний зарезервированы за устройствами, входящими в состав стандартной конфигурации ЭВМ, и на магистраль не выведены.

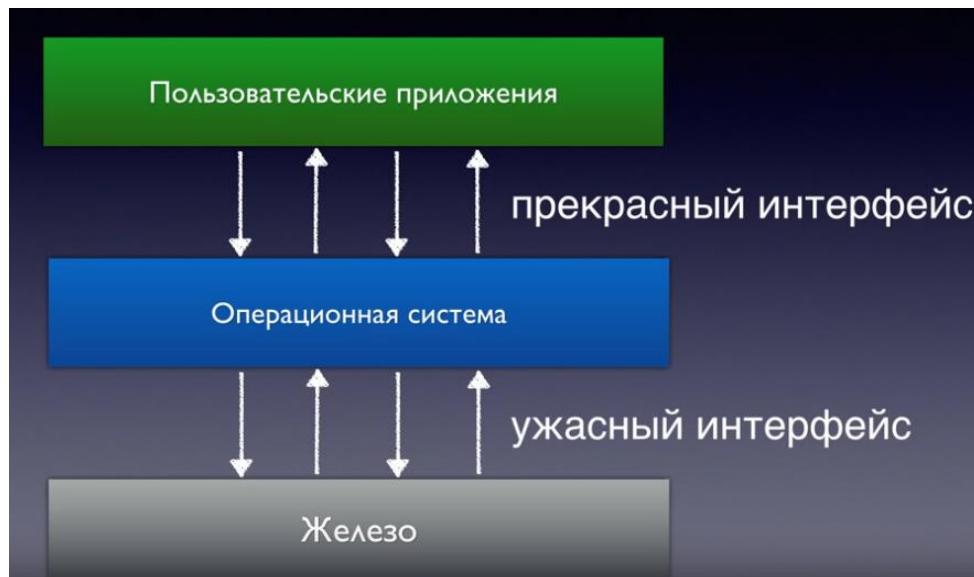
Хотя шина ISA имеет небольшую информационную ширину и в настоящее время используется для подключения только относительно медленных периферийных устройств, ее состав позволяет в определенной мере проследить взаимосвязь различных рассмотренных ранее устройств, составляющих ЭВМ.

Организация ЭВМ на основе общей шины является сдерживающим фактором для повышения производительности компьютера. Следует отметить, что даже при использовании прямого доступа к памяти процессор полностью не освобождается от управления операциями ввода-вывода. Он обеспечивает инициализацию контроллера ПДП, а также взаимодействует с ним по некоторым управляющим линиям. Более того, во время операции передачи данных интерфейс оказывается занятым, а связь процессора с оперативной памятью - блокированной.

Это существенно сказывается на эффективности работы ЭВМ, особенно в тех случаях, когда в вычислительной системе используется большое количество высокоскоростных внешних устройств. Для решения этой проблемы в состав высокопроизводительных компьютеров иногда включают специализированные процессоры ввода-вывода, способные полностью разгрузить основной процессор от управления операциями обмена с внешними устройствами.

## **4.5 Операционная система**

интерфейс взаимодействия ПО и железа. Это интерфейс, и абстракция



## ОС должна

- Управлять запуском нескольких процессов
- Предоставлять необходимые ресурсы процессу и защищать ресурсы процесса от остальных
- Позволять процессам обмениваться информацией
- Позволять процессам синхронизироваться

Бывают

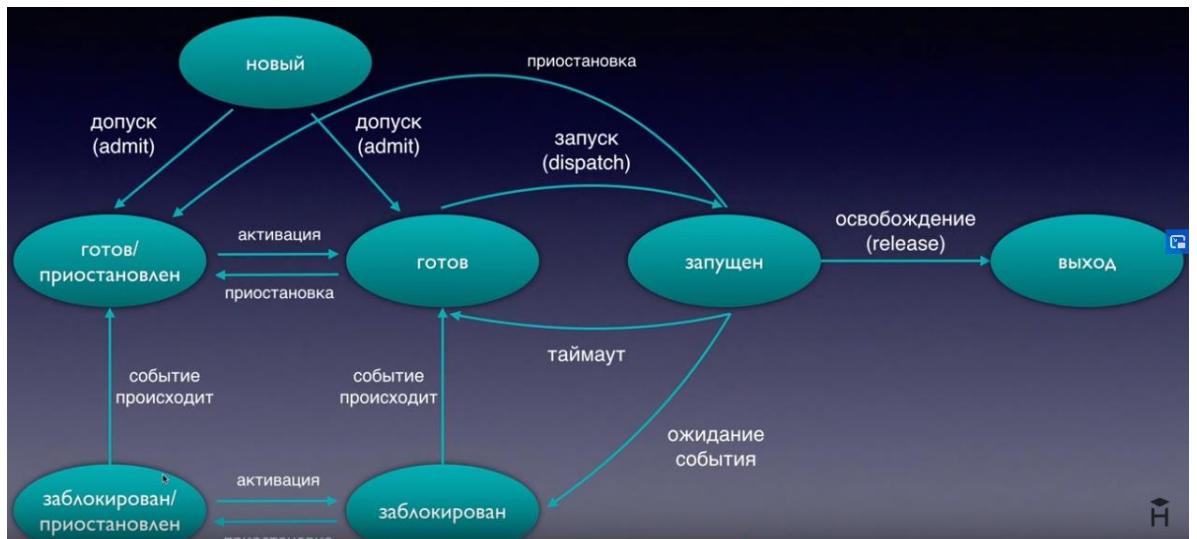
- Микроядра, в которых только самое важное (виртуальная память, планирование приложений), а драйвера, GUI (UNIX), файловая система и пр – в отдельных приложениях. Это хорошо, тк все яйца лежат по разным корзинам.
- Моноядро – это все в одном месте. Если крашнется – может крашнуться все (как было с Internet Explorer'ом).

#### 4.5.1 Процесс

## Процесс

- Идентификатор
- Состояние
- Приоритет
- Счетчик команд (Program counter)
- Указатели на память
- Контекст
- Информация о статусе I/O
- Другая информация

Пути работы



Причины остановки

- Swapping
- Решение ОС
- Решение пользователя
- Расписание
- Решение родительского процесса

#### 4.5.2 Очередь



#### 4.5.3 ОС – процесс

Вообще, процесс – это абстракция ОС. Бывает, что ОС реализованы по разному.

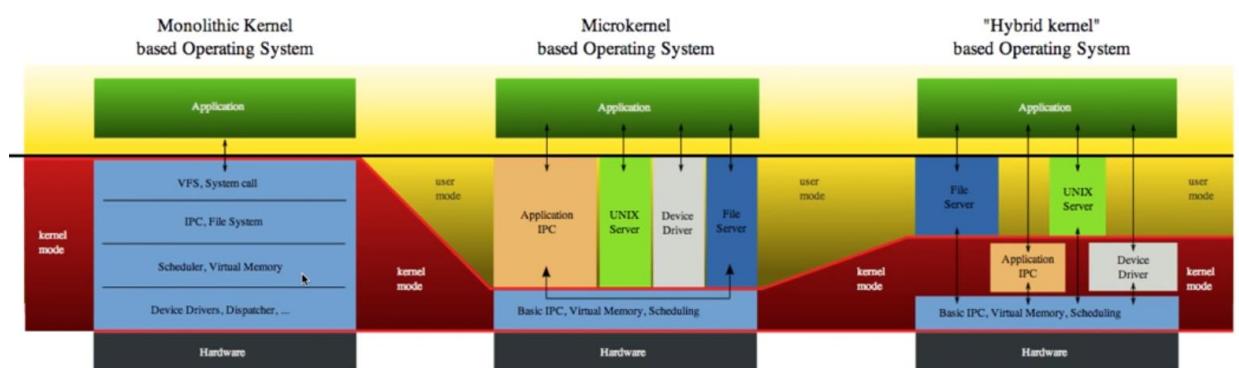
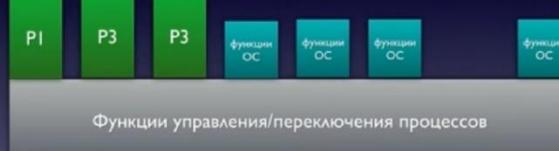


## Функции ОС в пользовательских процессах

- ОС предоставляет свои услуги “внутри” (в контексте) пользовательского процесса
- Не нужно переключаться для выполнения задач ОС



## Функции ОС как отдельные процессы



Хорошо, тк если функция написана плохо – ядро не сломается. Но ОС нужно будет переключать контекст, как и во всех процессах. И еще процессы не имеют доступ к железу, но функции ос могут требовать прямого доступа – нужно будет придумывать механизм, по которому процессу нужно давать доступ к железу.

Раньше я на XP получал доступ к LPT прошивание AVR контроллеров.

## 5

### 5.1 Стратегии управления оперативной памятью и Виртуальная память

Так как память является неотъемлемой частью компьютера, а при этом она медленнее ЦП, требуются стратегии управления ОП.

Хекслет, Операционные системы, урок 7: Организация памяти.  
Виртуальная память. <https://www.youtube.com/watch?v=x4FQ6ASzks0>

Задачи стратегий:

#### 5.1.1 Требования к стратегиям

##### 5.1.1.1 Распределение

- Программист не знает (и не должен знать) где в памяти будет находиться его приложение.
- Оно может быть временно перемещено на диск
- Обращения к памяти должны конвертироваться в настоящий физический адрес

##### 5.1.1.2 Защита

- Процессы не должны иметь возможности обратиться к области памяти, используемой другим процессом

##### 5.1.1.3 Разделение

- Доступ к одному участку памяти нескольким процессам
- Давать процессам доступ к одной и той же программе – эффективнее чем создавать копию программы для каждого процесса

#### *5.1.1.4 Логическая организация*

- Обычно память растет линейно
- Программы могут состоять из модулей, которые компилируются и запускаются отдельно друг от друга
- Разные модули могут иметь разные уровни доступа (только чтение, чтение и запись, и т.д.)
- Процессы могут использовать одни и те же модули

#### *5.1.1.5 Физическая организация*

- Программист не должен отвечать за физическую организацию
- Программист не знает сколько памяти доступно

### *5.1.2 Организация памяти*

#### *5.1.2.1 Единое постоянного использования*

## Single contiguous allocation

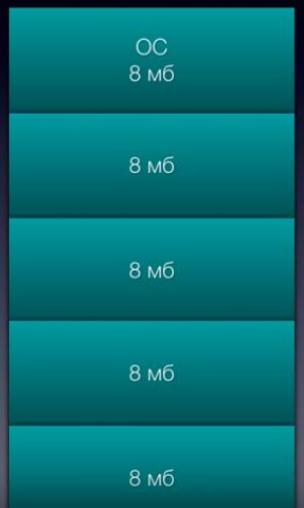
- Самый простой вариант организации памяти
- Вся память – одному процессу (кроме небольшого зарезервированного для ОС участка)
- MS-DOS
- Возможна ли многозадачность? Да, если есть swapping

### *5.1.3 Разделение памяти*

#### *5.1.3.1 Одинаковые участки*

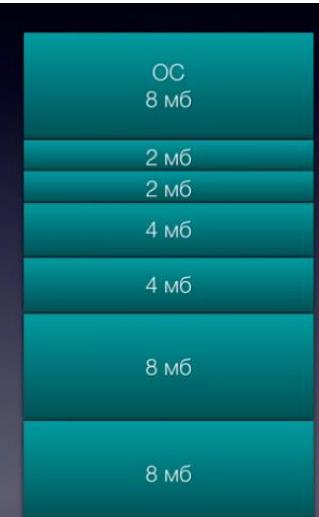
## Fixed partitioning

- Одинаковые по размеру разделы
- Любой процесс может занять раздел если помещается в него
- ОС может приостановить процесс и вынести раздел на вторичный носитель
- Программа должна умещаться в раздел
- Внутренняя фрагментация



#### *5.1.3.2 Неравные разделы*

- Чуть лучше, но проблемы остаются
- Всегда выбирать самый маленький возможный раздел
- Очередь для каждого раздела



### 5.1.3.3 Динамическое разделение

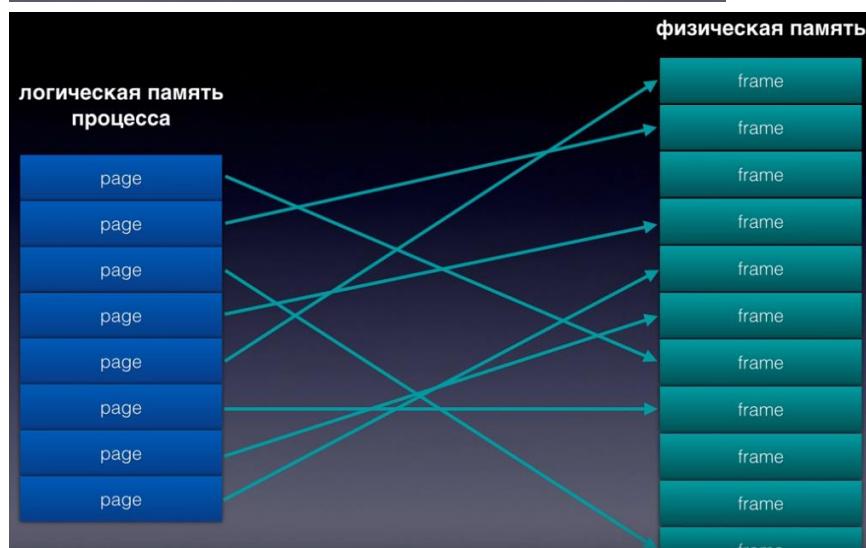
- Раздел нужного размера создается для каждого процесса
- Внешняя фрагментация
- При запуске процесса ОС должна решить какой пустой блок использовать для создания раздела
  - Best fit: выбрать самый близкий по размеру блок
  - First fit: выбрать первый подходящий блок
  - Next fit: выбрать следующий подходящий блок после последнего выбора

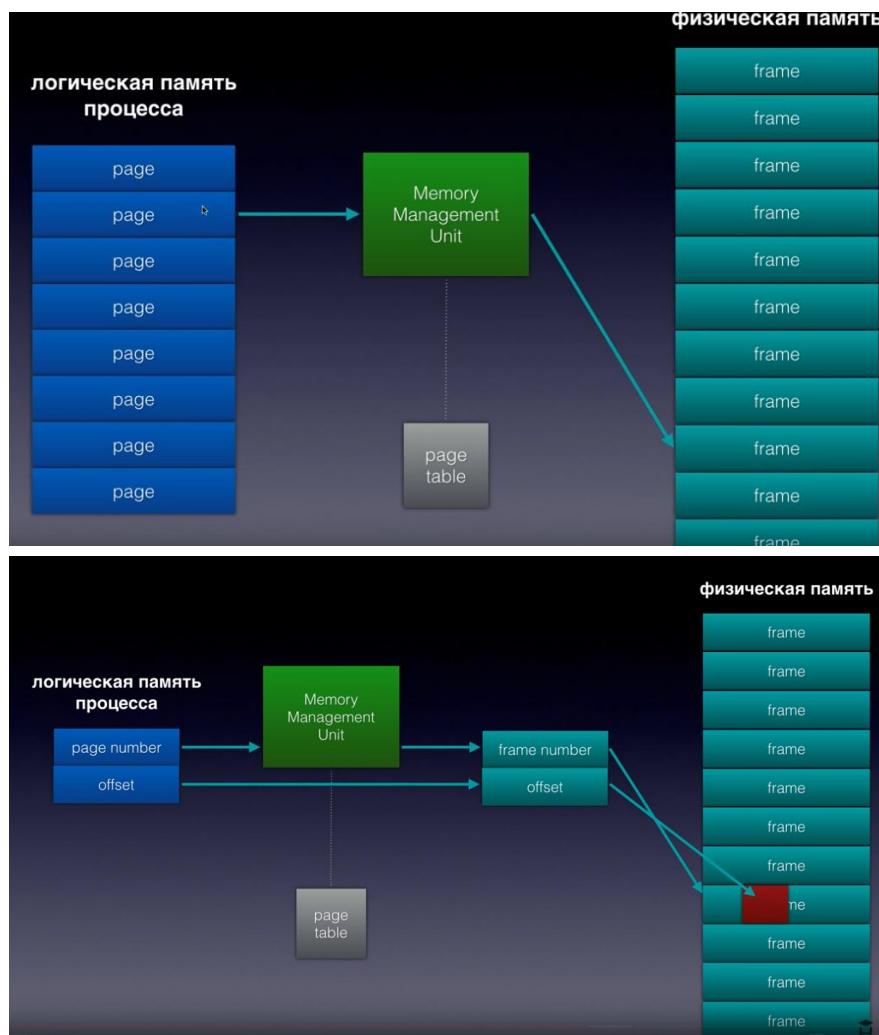


### 5.1.3.4 Подкачка страниц

(paging)

- Страница (page) – участок памяти стандартного размера
- Процессы видят память постранично
- Каждый процесс видит свой приватный набор страниц со своей логической адресацией
- При обращении к памяти ОС конвертирует логический адрес в абсолютный





Если фрейм оказался на вторичном хранилище – возникает page fault

- Когда процесс обращается к памяти, которая физически находится во вторичном хранилище
- Memory Management Unit (MMU) вызывает прерывание
- ОС загружает страницу в память (возможно, выгружая что-то оттуда на диск)

### *5.1.3.5 Виртуальная память*

## Виртуальная память

- Процессы обычно не используют всю доступную им память
- Структуры данных часто имеют избыточный размер
- Части используемой памяти могут использоваться в разное время
- Временный перенос страниц на вторичный носитель может улучшить ситуацию
- Логическое адресное пространство для процесса может превышать по размеру первичную память!

Часть фреймов лежит на вторичной памяти

### *5.1.3.6 Подгрузка фреймов*

- Ленивая (только тогда, когда нужно)
- Пред загрузка след страниц

### *5.1.3.7 Copy on write*

- В Unix процессы создаются с помощью fork – клонирования существующего процесса
- Копирование памяти при клонировании процесса может занимать много времени
- Решение: в начале использовать одну и ту же память для старого и нового процесса; копировать память только когда необходима запись.



#### 5.1.4 Пробуксовка

## Thrashing

- Перенос блоков памяти во вторичное хранилище и обратно – дорогая операция
- ОС может тратить слишком много времени на эти операции
- Предзагрузка страниц может помочь

Возможна коллизия данных в кэше

## 5.2 Статическая и динамическая сборка

Библиотеки предварительно компилируют по нескольким причинам. Во-первых, их код редко меняется. Было бы напрасной тратой времени повторно компилировать библиотеку каждый раз при её использовании в новой программе. Во-вторых, поскольку весь код предварительно скомпилирован в машинный язык, то это предотвращает получение доступа к исходному коду (и его изменение) сторонними лицами. Этот пункт важен для предприятий/людей, которые не хотят, чтобы результат их труда (исходный код) был доступен всем (интеллектуальная собственность, все дела).

Есть два типа библиотек: статические и динамические.

### 5.2.1 Статическая библиотека

\*.lib / \*.a

Статическая библиотека (или ещё «архив») состоит из подпрограмм, которые непосредственно компилируются и линкуются с вашей программой. При компиляции программы, которая использует статическую библиотеку, весь функционал статической библиотеки (тот, что использует ваша программа) становится частью вашего исполняемого файла. В Windows статические библиотеки имеют расширение .lib (от «library»), тогда как в Linux статические библиотеки имеют расширение .a (от «archive»).

Одним из преимуществ статических библиотек является то, что вам нужно распространять всего лишь 1 файл (исполнимый файл), дабы пользователи могли запустить и использовать вашу программу. Поскольку статические библиотеки становятся частью вашей программы, то вы можете использовать их подобно функционалу своей собственной программы. С другой стороны, поскольку копия библиотеки становится частью каждого вашего исполняемого файла, то это может привести к увеличению размера файла. Также, если вам нужно будет обновить статическую библиотеку, вам придётся перекомпилировать каждый исполняемый файл, который её использует.

### *5.2.2 Динамическая библиотека*

\*.dll / \*.so

Динамическая библиотека (или ещё «общая библиотека») состоит из подпрограмм, которые подгружаются в вашу программу во время её выполнения. При компиляции программы, которая использует динамическую библиотеку, эта библиотека не становится частью вашего исполняемого файла — она так и остаётся отдельным модулем. В Windows динамические библиотеки имеют расширение .dll (от «dynamic link library» = «библиотека динамической компоновки»), тогда как в Linux динамические библиотеки имеют расширение .so (от «shared object» = «общий объект»). Одним из преимуществ динамических библиотек является то, что разные программы могут совместно использовать одну копию динамической библиотеки, что значительно экономит используемое пространство. Ещё одним преимуществом динамической библиотеки является то, что её можно обновлять до более новой версии без перекомпиляции всех исполняемых файлов, которые её используют.

Поскольку динамические библиотеки не линкуются непосредственно с вашей программой, то ваши программы, использующие динамические библиотеки, должны явно подключать и взаимодействовать с динамической библиотекой. Этот механизм не всегда может быть понятен для новичков, что

может затруднить взаимодействие с динамической библиотекой. Для упрощения этого процесса используют библиотеки импорта.

### *5.2.3 Библиотека импорта*

Библиотека импорта (англ. «import library») — это библиотека, которая автоматизирует процесс подключения и использования динамической библиотеки. В Windows это обычно делается через небольшую статическую библиотеку (.lib) с тем же именем, что и динамическая библиотека (.dll). Статическая библиотека линкуется с вашей программой во время компиляции, и тогда функционал динамической библиотеки может эффективно использоваться в вашей программе, как если бы это была обычная статическая библиотека. В Linux общий объектный файл (с расширением .so) дублируется сразу как динамическая библиотека и библиотека импорта. Большинство линкеров при создании динамической библиотеки автоматически создают к ней библиотеку импорта.

### *5.2.4 Короткое объяснение*

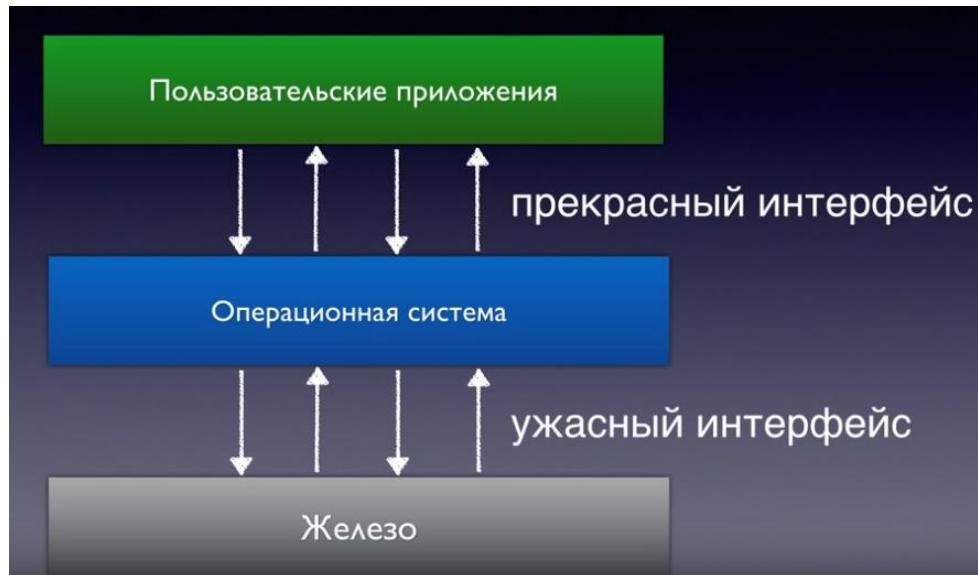
Когда Вы собираете Ваш проект, и хотите включить в него библиотеку(собранную Статически или Динамически \*.a или \*.so) происходит связывание ld всего Вашего кода. Когда вы где-то пишите, что тут будет вызываться функция библиотеки А, компилятор оставляет там пометку (по сути обещание), что референс на данные call будет подставлен на этапе линковки. Далее ликвидатор смотрит на флаги связывания SHARED или STATIC(что и отвечает за динамическую или статическую библиотеку) и ищет ее согласно стандартным путям и/или указанным Вами путям.

Статическая библиотека - (\*.a) собрана для непосредственного встраивания в Ваш исполняемый файл. Она просто будет помещена в соответствии с указанием linker'a. Тут будет статическая линковка.

Динамическая библиотека - (\*.so) - будет просто подключаться как link на референс и не попадет в Ваш бинарь. Будет лишь указание где брать референс на тот или иной функционал. Тут будет динамическая линковка.

## 6.1 Распределение и использование ресурсов вычислительной системы и управление ими.

интерфейс взаимодействия ПО и железа. Это интерфейс, и абстракция



Управление ресурсами вычислительной системы с целью наиболее эффективного их использования является назначением операционной системы. Например, мультипрограммная операционная система организует одновременное выполнение сразу нескольких процессов на одном компьютере, поочерёдно переключая процессор с одного процесса на другой, исключая простой процессора, вызываемые обращениями процессов к вводу-выводу ОС также отслеживает и разрешает конфликты, возникающие при обращении нескольких процессов к одному и тому же устройству ввода-вывода или к одним и тем же данным.

Критерий эффективности, в соответствии с которым ОС организует управление ресурсами компьютера, может быть различным. Например, в одних системах важен такой критерий, как пропускная способность вычислительной системы, в других — время её реакции. Соответственно выбранному критерию эффективности операционные системы по-разному организуют вычислительный процесс.

Управление ресурсами включает решение следующих общих, не зависящих от ресурса задач:

- планирование ресурса, то есть определение, какому процессу, когда и в каком количестве (если ресурс может выделяться частями) следует выделить ресурс;
- удовлетворение запросов на ресурсы;
- отслеживание состояния и учёт использования ресурса, то есть поддержание оперативной информации о том, занят или свободен ресурс и какая доля ресурса уже распределена;
- разрешение конфликтов между процессами.

Задача организации эффективного совместного использования ресурсов несколькими процессами является весьма сложной, и сложность эта порождается в основном случайным характером возникновения запросов на потребление ресурсов. В мультипрограммной системе образуются очереди заявок от одновременно выполняемых программ к разделяемым ресурсам компьютера: процессору, страницам памяти, к принтерам, к дискам. Операционная система организует обслуживание этих очередей по разным алгоритмам: в порядке поступления, на основе приоритетов, кругового обслуживания и т. д.

Планирование очень сильно влияет на общую производительность.

Если для решения очередной задачи не хватает ресурсов, ОС должна принять одно из следующих решений:

- отобрать часть ресурсов у какой - либо другой задачи, выполнявшейся в данный момент и менее приоритетной;
- подождать, пока какая-нибудь из решаемых задач завершится и освободит требуемый ресурс;
- пропустить вне очереди ту задачу, чья очередь еще не подошла, но для выполнения которой ресурсов достаточно.

# Виды планирования

- Долгосрочное: добавление процесса в общий пул для будущего запуска
- Среднесрочное: добавление процесса в память
- Краткосрочное: выбор процесса для непосредственного запуска на ЦП
- I/O: какие из ожидающих запросов должны быть обслужены устройством ввода-вывода

## *6.1.1 Долгосрочное планирование*

- Может работать по принципу first-come-first-served
- Может использовать приоритеты
- Влияет на общее количество процессов

## *6.1.2 Среднесрочное планирование*

- Swapping (приостановка процесса и помещение его во вторичное хранилище)

## *6.1.3 Краткосрочное планирование*

- Dispatcher
- Запускается чаще всех
- Главная задача – эффективное использование ресурса ЦП

Dispatcher – утилита в ОС, которая занимается запускателем программ. Обычно это лучшее инженерное творение в ОС, тк запускается чаще всех.

- Для пользователя: нужно сократить время отклика
- Для системы: нужно эффективно использовать ЦП

## 6.2 Основные подходы и алгоритмы планирования.

- **Turnaround time** (время полного оборота): время, прошедшее с момента отправки процесса в пул до его завершения.
- **Response time** (время отклика): время, прошедшее с момента отправки запроса процессу до того, как ответ начнет возвращаться.
- **Throughput** (производительность): количество процессов, завершающих свою работу за отрезок времени.
- **Process utilization** (использование ЦП): процент времени когда ЦП используется.

### 6.2.1 Приоритеты

- Процессы могут иметь разные приоритеты.
- В первую очередь обслуживаются процессы с высшим приоритетом.

# Проблема приоритетов: голодание

- Проблема: процессы низкого приоритета могут бесконечно долго ждать своей очереди.
- Решение: повышать приоритет процесса в зависимости от его возраста.

## 6.2.2 Функция выбора

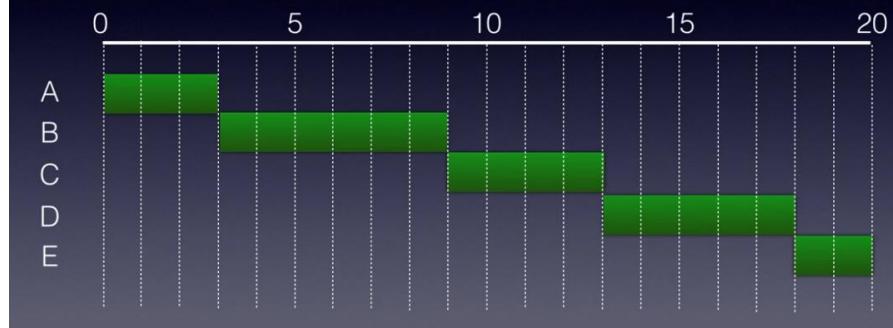
- Функция, определяющая какой процесс будет запущен следующим
- Может использовать следующие показатели:
  - w – сколько процесс ждал
  - e – сколько процесс работал (был запущен)
  - s – сколько времени необходимо процессу для завершения

### Пример

Процесс	Время появления	Время работы
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

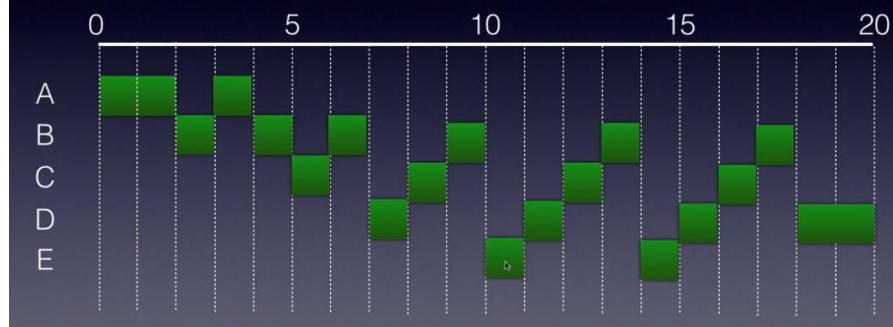
#### 6.2.2.1 Первый зашел – первый стал обслужен

##### First-come-first-served



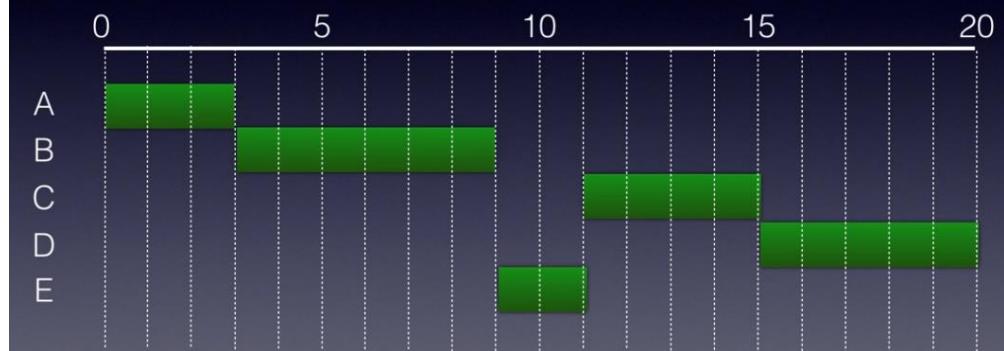
#### 6.2.2.2 Циклический

##### Round robin



#### 6.2.2.3 Следующий - кратчайший

##### Shortest Process Next



- Сложно предугадать время выполнения долгих процессов

- Возможное ресурсное голодание для долгих процессов

#### *6.2.2.4 Следующий – тот, кому осталось меньше времени*

### **6.3 Системы реального и разделенного времени.**

#### *6.3.1 Системы реального времени*

Операционная система реального времени (ОСРВ, англ. real-time operating system, RTOS) — тип операционной системы, основное назначение которой — предоставление необходимого и достаточного набора функций для проектирования, разработки и функционирования систем реального времени на конкретном аппаратном оборудовании.

Спецификация UNIX в редакции 2 даёт следующее определение: Реальное время в операционных системах — это способность операционной системы обеспечить требуемый уровень сервиса в определённый промежуток времени.

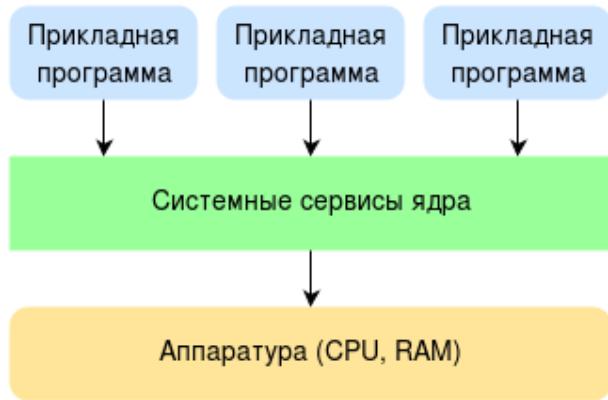
Идеальная ОСРВ имеет предсказуемое поведение при всех сценариях нагрузки, включая одновременные прерывания и выполнение потоков

	ОС реального времени	ОС общего назначения
Основная задача	Успеть среагировать на события, происходящие на оборудовании	Оптимально распределить ресурсы компьютера между пользователями и задачами
На что ориентирована	Обработка внешних событий	Обработка действий пользователя
Как позиционируется	Инструмент для создания конкретного аппаратно-программного комплекса реального времени	Воспринимается пользователем как набор приложений, готовых к использованию
Кому предназначена	Квалифицированный разработчик	Пользователь средней квалификации

#### *6.3.2 Архитектуры*

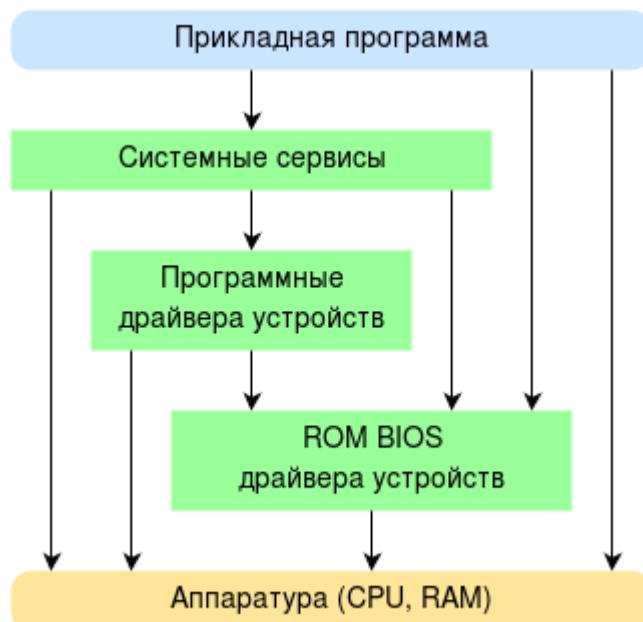
##### *6.3.2.1 Монолитная архитектура.*

ОС определяется как набор модулей, взаимодействующих между собой внутри ядра системы и предоставляющих прикладному ПО входные интерфейсы для обращений к аппаратуре. Основной недостаток этого принципа построения ОС заключается в плохой предсказуемости её поведения, вызванной сложным взаимодействием модулей между собой.



### 6.3.2.2 Уровневая (слоевая) архитектура.

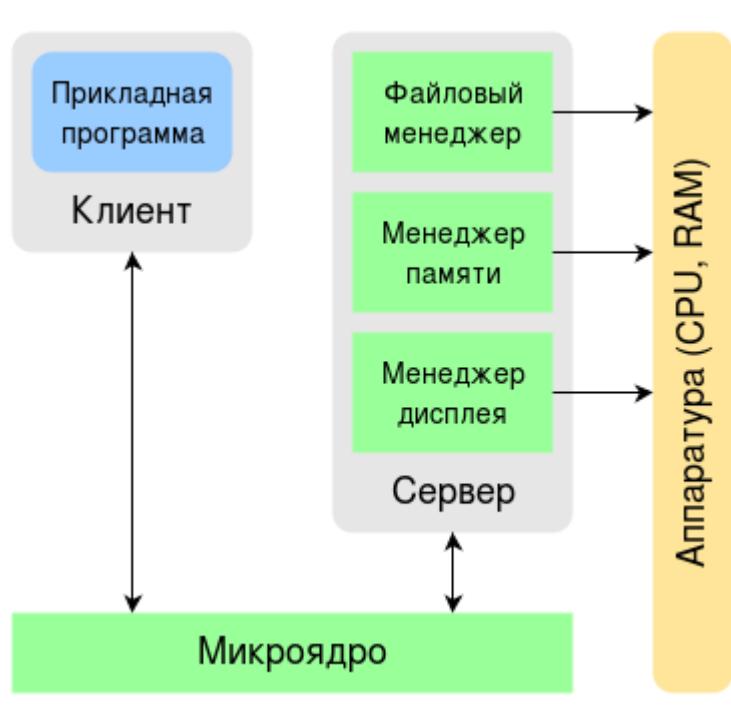
Прикладное ПО имеет возможность получить доступ к аппаратуре не только через ядро системы и её сервисы, но и напрямую. По сравнению с монолитной такая архитектура обеспечивает значительно большую степень предсказуемости реакций системы, а также позволяет осуществлять быстрый доступ прикладных приложений к аппаратуре. Главным недостатком таких систем является отсутствие многозадачности.



### 6.3.2.3 Архитектура «клиент-сервер»

Основной её принцип заключается в вынесении сервисов ОС в виде серверов на уровень пользователя и выполнении микроядром функций диспетчера сообщений между клиентскими пользовательскими программами и серверами — системными сервисами. Преимущества такой архитектуры:

- Повышенная надёжность, так как каждый сервис является, по сути, самостоятельным приложением и его легче отладить и отследить ошибки.
- Улучшенная [масштабируемость](#), поскольку ненужные сервисы могут быть исключены из системы без ущерба к её работоспособности.
- Повышенная отказоустойчивость, так как «зависший» сервис может быть перезапущен без перезагрузки системы.



### 6.3.3 Особенности ядра

Ядро ОСРВ обеспечивает функционирование промежуточного абстрактного уровня ОС, который скрывает от прикладного ПО специфику технического устройства процессора (нескольких процессоров) и связанного с ним аппаратного обеспечения

Указанный абстрактный уровень предоставляет для прикладного ПО пять основных категорий сервисов:

- Управление задачами. Самая главная группа сервисов. Позволяет разработчикам приложений проектировать программные продукты в виде наборов отдельных программных фрагментов, каждый из которых может относиться к своей тематической области, выполнять отдельную функцию и иметь свой собственный квант времени, отведенный ему для работы. Каждый такой фрагмент называется задачей. Сервисы в рассматриваемой группе обладают способностью запускать задачи и присваивать им приоритеты. Основной сервис здесь — планировщик

задач. Он осуществляет контроль над выполнением текущих задач, запускает новые в соответствующий период времени и следит за режимом их работы.

- Динамическое распределение памяти. Многие (но не все) ядра ОСРВ поддерживает эту группу сервисов. Она позволяет задачам заимствовать области оперативной памяти для временного использования в работе приложений. Часто эти области впоследствии переходят от задачи к задаче, и посредством этого осуществляется быстрая передача большого количества данных между ними. Некоторые очень малые по размеру ядра ОСРВ, которые предполагается использовать в аппаратных средах со строгим ограничением на объём используемой памяти, не поддерживают сервисы динамического распределения памяти.
- Управление таймерами. Так как встроенные системы предъявляют жёсткие требования к временным рамкам выполнения задач, в состав ядра ОСРВ включается группа сервисов, обеспечивающих управление таймерами для отслеживания лимита времени, в течение которого должна выполняться задача. Эти сервисы измеряют и задают различные промежутки времени (от 1 мкс и выше), генерируют прерывания по истечении временных интервалов и создают разовые и циклические будильники.
- Взаимодействие между задачами и синхронизация. Сервисы данной группы позволяют задачам обмениваться информацией и обеспечивают её сохранность. Они также дают возможность программным фрагментам согласовывать между собой свою работу для повышения эффективности. Если исключить эти сервисы из состава ядра ОСРВ, то задачи начнут обмениватьсяискаженной информацией и могут стать помехой для работы соседних задач.
- Контроль устройства ввода-вывода. Сервисы этой группы обеспечивают работу единого программного интерфейса, взаимодействующего со всем множеством драйверов устройств, которые являются типичными для большинства встроенных систем.

В дополнение к сервисам ядра, многие ОСРВ предлагают линейки дополнительных компонентов для организации таких высокоуровневых понятий, как файловая система, сетевое взаимодействие, управление сетью, управление базой данных, графический пользовательский интерфейс и т. д. Хотя

многие из этих компонентов намного больше и сложнее, чем само ядро ОСРВ, они, тем не менее, основываются на его сервисах. Каждый из таких компонентов включается во встроенную систему, только если её сервисы необходимы для выполнения встроенного приложения и только для того, чтобы свести расход памяти к минимуму[7].

#### *6.3.4 Отличия от операционных систем общего назначения*

Многие операционные системы общего назначения также поддерживают указанные выше сервисы. Однако ключевым отличием сервисов ядра ОСРВ является *детерминированный*, основанный на строгом контроле времени, характер их работы. В данном случае под детерминированностью понимается то, что для выполнения одного сервиса операционной системы требуется временной интервал, заведомо известной продолжительности. Теоретически это время может быть вычислено по математическим формулам, которые должны быть строго алгебраическими и не должны включать никаких временных параметров случайного характера. Любая случайная величина, определяющая время выполнения задачи в ОСРВ, может вызвать нежелательную задержку в работе приложения, тогда следующая задача не уложится в свой квант времени, что послужит причиной для ошибки.

В этом смысле операционные системы общего назначения не являются детерминированными. Их сервисы могут допускать случайные задержки в своей работе, что может привести к замедлению ответной реакции приложения на действия пользователя в заведомо неизвестный момент времени. При проектировании обычных операционных систем разработчики не акцентируют своё внимание на математическом аппарате вычисления времени выполнения конкретной задачи и сервиса. Это не является критичным для подобного рода систем.

#### *6.3.5 Работа планировщика*

Большинство ОСРВ выполняет планирование задач, руководствуясь следующей схемой[7]. Каждой задаче в приложении ставится в соответствие некоторый приоритет. Чем больше приоритет, тем выше должна быть реактивность задачи. Высокая реактивность достигается путём реализации подхода приоритетного вытесняющего планирования (*preemptive priority scheduling*), суть которого заключается в том, что планировщику разрешается

останавливать выполнение любой задачи в произвольный момент времени, если установлено, что другая задача должна быть запущена незамедлительно.

Описанная схема работает по следующему правилу: если две задачи одновременно готовы к запуску, но первая обладает высоким приоритетом, а вторая — низким, то планировщик отдаст предпочтение первой. Вторая задача будет запущена только после того, как завершит свою работу первая.

Возможна ситуация, когда задача с низким приоритетом уже запущена, а планировщик получает сообщение, что другая задача с более высоким приоритетом готова к запуску. Причиной этому может послужить какое-либо внешнее воздействие (прерывание от оборудования), как, например, изменение состояния переключателя устройства, управляемого ОСРВ. В такой ситуации планировщик задач поведет себя согласно подходу приоритетного вытесняющего планирования следующим образом. Задаче с низким приоритетом будет позволено выполнить до конца текущую машинную команду (но не команду, описанную в исходнике программы языком высокого уровня), после чего выполнение задачи приостанавливается[7]. Далее запускается задача с высоким приоритетом. После того, как она прорабатывает, планировщик запускает прерванную первую задачу с машинной команды, следующей за последней выполненной.

Каждый раз, когда планировщик задач получает сигнал о наступлении некоторого внешнего события (триггер), причина которого может быть как аппаратная, так и программная, он действует по следующему алгоритму[7]:

- Определяет, должна ли текущая выполняемая задача продолжать работать.
- Устанавливает, какая задача должна запускаться следующей.
- Сохраняет контекст остановленной задачи (чтобы она потом возобновила работу с места остановки).
- Устанавливает контекст для следующей задачи.
- Запускает эту задачу.

### *6.3.6 Выполнение задачи*

В обычных ОСРВ задача может находиться в трёх возможных состояниях:

- задача выполняется;
- задача готова к выполнению;
- задача заблокирована.

Большую часть времени основная масса задач заблокирована. Только одна задача может выполняться на центральном процессоре в текущий момент времени. В примитивных ОСРВ список готовых к исполнению задач, как правило, очень короткий, он может состоять не более чем из двух-трёх наименований.

Основная функция администратора ОСРВ заключается в составлении такого планировщика задач.

Если в списке готовых к выполнению задач последних имеется не больше двух—трёх, то предполагается, что все задачи расположены в оптимальном порядке. Если же случаются такие ситуации, что число задач в списке превышает допустимый лимит, то задачи сортируются в порядке приоритета.

### *6.3.7 Алгоритмы планирования*

В настоящее время для решения задачи эффективного планирования в ОСРВ наиболее интенсивно развиваются два подхода[9]:

- статические алгоритмы планирования (RMS, Rate Monotonic Scheduling) — используют приоритетное вытесняющее планирование, приоритет присваивается каждой задаче до того, как она начала выполняться, преимущество отдаётся задачам с самыми короткими периодами выполнения;
- динамические алгоритмы планирования (EDF, Earliest Deadline First Scheduling) — приоритет задачам присваивается динамически, причём предпочтение отдаётся задачам с наиболее ранним предельным временем начала (завершения) выполнения.

При больших загрузках системы EDF более эффективен, нежели RMS.

### *6.3.8 Взаимодействие между задачами и разделение ресурсов*

Многозадачным системам необходимо распределять доступ к ресурсам. Одновременный доступ двух и более процессов к какой-либо области памяти или другим ресурсам представляет определённую угрозу. Существует три способа решения этой проблемы: временное блокирование прерываний, двоичные семафоры, посылка сигналов. ОСРВ обычно не используют первый способ, потому что пользовательское приложение не может контролировать процессор столько, сколько хочет. Однако во многих встроенных системах и ОСРВ позволяет запускать приложения в режиме ядра для доступа к системным вызовам и даётся контроль над окружением исполнения без вмешательства ОС.

На однопроцессорных системах наилучшим решением является приложение, запущенное в режиме ядра, которому позволено блокирование прерываний. Пока прерывание заблокировано, приложение использует ресурсы процесса единолично и никакая другая задача или прерывание не может выполняться. Таким образом защищаются все критичные ресурсы. После того как приложение завершит критические действия, оно должно разблокировать прерывания, если таковые имеются. Временное блокирование прерывания позволено только тогда, когда самый долгий промежуток выполнения критической секции меньше, чем допустимое время реакции на прерывание. Обычно этот метод защиты используется, только когда длина критического кода не превышает нескольких строк и не содержит циклов. Этот метод идеально подходит для защиты регистров.

Когда длина критического участка больше максимальной или содержит циклы, программист должен использовать механизмы, идентичные или имитирующие поведение систем общего назначения, такие, как семафоры и посылка сигналов.

### *6.3.9 Выделение памяти*

Следующим проблемам выделения памяти в ОСРВ уделяется больше внимания, нежели в операционных системах общего назначения.

Во-первых, скорости выделения памяти. Стандартная схема выделения памяти предусматривает сканирование списка неопределенной длины для нахождения свободной области памяти заданного размера, а это неприемлемо, так как в ОСРВ выделение памяти должно происходить за фиксированное время.

Во-вторых, память может стать фрагментированной в случае разделения свободных её участков уже запущенными процессами. Это может привести к остановке программы из-за её неспособности задействовать новый участок памяти. Алгоритм выделения памяти, постепенно увеличивающий фрагментированность памяти, может успешно работать на настольных системах, если те перезагружаются не реже одного раза в месяц, но является неприемлемым для встроенных систем, которые работают годами без перезагрузки.

Простой алгоритм с фиксированной длиной участков памяти очень хорошо работает в несложных встроенных системах.

Также этот алгоритм отлично функционирует и в настольных системах, особенно тогда, когда во время обработки участка памяти одним ядром

следующий участок памяти обрабатывается другим ядром. Такие оптимизированные для настольных систем ОСРВ, как Unison Operating System или DSPnano RTOS, предоставляют указанную возможность.

## 7

### 7.1 Взаимодействие процессов.

Межпроцессное взаимодействие (англ. inter-process communication, IPC) — обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

Из механизмов, предоставляемых ОС и используемых для IPC, можно выделить:

- механизмы обмена сообщениями;
- механизмы синхронизации;
- механизмы разделения памяти;
- механизмы удалённых вызовов (RPC).

Для оценки производительности различных механизмов IPC используют следующие параметры:

- пропускная способность (количество сообщений в единицу времени, которое ядро ОС или процесс способно обработать);
- задержки (время между отправкой сообщения одним потоком и его получением другим потоком).

IPC может называться терминами межпотоковое взаимодействие (англ. inter-thread communication) и межпрограммное взаимодействие (англ. inter-application communication).

Методы межпроцессного взаимодействия описаны ниже.

Метод	Реализуется ОС или процессом
Файл	Все ОС.
Сигнал	Большинство ОС; в некоторых ОС, например, в Windows, сигналы доступны только в библиотеках, реализующих стандартную библиотеку языка Си, и не могут использоваться для IPC.
Сокет	Большинство ОС.
Канал	Все ОС, совместимые со стандартом POSIX.
Именованный канал	Все ОС, совместимые со стандартом POSIX.
Неименованный канал	Все ОС, совместимые со стандартом POSIX.
Семафор	Все ОС, совместимые со стандартом POSIX.
Разделяемая память	Все ОС, совместимые со стандартом POSIX.
Обмен сообщениями (без разделения)	Используется в парадигме MPI, Java RMI, CORBA и других.
Проецируемый в память файл (mmap)	Все ОС, совместимые со стандартом POSIX. При использовании временного файла возможно возникновение гонки. ОС Windows также предоставляет этот механизм, но посредством API, отличающегося от API, описанного в стандарте POSIX.
Очередь сообщений (Message queue)	Большинство ОС.
Почтовый ящик	Некоторые ОС.

### 7.1.1 Файл

### 7.1.2 Сигнал

Сигнал в операционных системах семейства Unix — асинхронное уведомление процесса о каком-либо событии, один из основных способов взаимодействия между процессами. Когда сигнал послан процессу, операционная система прерывает выполнение процесса, при этом, если процесс установил собственный обработчик сигнала, операционная система запускает этот обработчик, передав ему информацию о сигнале, если процесс не установил обработчик, то выполняется обработчик по умолчанию.

Названия сигналов «SIG...» являются числовыми константами (макроопределениями Си) со значениями, определяемыми в заголовочном файле signal.h. Числовые значения сигналов могут меняться от системы к системе, хотя основная их часть имеет в разных системах одни и те же значения. Утилита kill позволяет задавать сигнал как числом, так и символьным обозначением.

Спецификация сигналов включена в стандарты POSIX.

### 7.1.3 POSIX

POSIX (англ. Portable Operating System Interface — переносимый интерфейс операционных систем) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка С и набор приложений и их интерфейсов. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix систем.

Серия стандартов POSIX была разработана комитетом 1003 IEEE. Международная организация по стандартизации (ISO) совместно с Международной электротехнической комиссией (IEC) приняли стандарт POSIX под названием ISO/IEC 9945[2]. Версии стандарта POSIX являются основой соответствующих версий стандарта Single UNIX Specification. Стандарт POSIX определяет интерфейс операционной системы, а соответствие стандарту Single UNIX Specification определяет реализацию интерфейса и позволяет операционным системам использовать торговую марку UNIX[3].

Название «POSIX» было предложено Ричардом Столлманом[4]. Введение в POSIX.1 гласит: «Ожидается произношение „позикс“ как в слове „позитив“, а не „посикс“. Произношение опубликовано в целях обнародования стандартного способа ссылки на стандартный интерфейс операционной системы». «POSIX» является зарегистрированным товарным знаком IEEE[4].

#### *7.1.4 UNIX*

Unix («UNIX» является зарегистрированной торговой маркой организации The Open Group[1]) — семейство переносимых, многозадачных и многопользовательских операционных систем, которые основаны на идеях оригинального проекта AT&T Unix, разработанного в 1970-х годах в исследовательском центре Bell Labs Кеном Томпсоном, Деннисом Ритчи и другими.

Операционные системы семейства Unix характеризуются модульным дизайном, в котором каждая задача выполняется отдельной утилитой, взаимодействие осуществляется через единую файловую систему, а для работы с утилитами используется командная оболочка.

#### *7.1.5 Сокет*

Сокет (англ. socket — разъём) — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Для взаимодействия между машинами с помощью стека протоколов TCP/IP используются адреса и порты. Адрес представляет собой 32-битную структуру для протокола IPv4, 128-битную для IPv6. Номер порта — целое число в диапазоне от 0 до 65535 (для протокола TCP).

Эта пара определяет сокет («гнездо», соответствующее адресу и порту).

Типы сокетов:

- Поточный - обеспечивает двухсторонний, последовательный, надежный, и недублированный поток данных без определенных границ. Тип сокета - SOCK\_STREAM, в домене Интернета он использует протокол TCP.
- Датаграммный- поддерживает двухсторонний поток сообщений. Приложение, использующее такие сокеты, может получать сообщения в порядке, отличном от последовательности, в которой эти сообщения посылались. Тип сокета - SOCK\_DGRAM, в домене Интернета он использует протокол UDP.
- Сокет последовательных пакетов - обеспечивает двухсторонний, последовательный, надежный обмен датаграммами фиксированной максимальной длины. Тип сокета - SOCK\_SEQPACKET. Для этого типа сокета не существует специального протокола.
- Простой сокет - обеспечивает доступ к основным протоколам связи.

### *7.1.6 Именованный канал*

В программировании именованный канал или именованный конвейер (англ. named pipe) — один из методов межпроцессного взаимодействия, расширение понятия конвейера в Unix и подобных ОС. Именованный канал позволяет различным процессам обмениваться данными, даже если программы, выполняющиеся в этих процессах, изначально не были написаны для взаимодействия с другими программами. Это понятие также существует и в Microsoft Windows, хотя там его семантика существенно отличается. Традиционный канал — «безымянен», потому что существует анонимно и только во время выполнения процесса. Именованный канал — существует в системе и после завершения процесса. Он должен быть «отсоединен» или удалён, когда уже не используется. Процессы обычно подсоединяются к каналу для осуществления взаимодействия между ними.

### *7.1.7 Канал*

См предыдущий. В терминологии операционных систем семейства Unix — некоторое множество процессов, для которых выполнено следующее перенаправление ввода-вывода: то, что выводит на поток стандартного вывода

предыдущий процесс, попадает в поток стандартного ввода следующего процесса.

### 7.1.8 Семафор

Примитив синхронизации[1] работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующейся[2]. Служит для построения более сложных механизмов синхронизации[1] и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.

Но есть проблема DEADLOCK:

- Запретить множественный лок на уровне ОС. Проблема, что это может мешать решать потребительские задачи
- Объединять несколько ресурсов в один лок. Проблема, что залочив мы можем использовать не все ресурсы, а только часть.

Семафоры могут быть двоичными и вычислительными[3]. Вычислительные семафоры могут принимать целочисленные неотрицательные значения и используются для работы с ресурсами, количество которых ограничено[3], либо участвуют в синхронизации параллельно исполняемых задач. Двоичные семафоры могут принимать только значения 0 и 1[3] и используются для взаимного исключения одновременного нахождения двух или более процессов в своих критических областях[4].

Мьютексные семафоры[3] или мьютексы — упрощённая реализация семафоров, аналогичная двоичным семафорам с тем отличием, что мьютексы должны отпускаться тем же процессом или потоком, который осуществляет их захват[5]. Наряду с двоичными семафорами используются в организации критических участков кода. В отличие от двоичных семафоров, начальное состояние мьютекса не может быть захваченным и они могут поддерживать наследование приоритетов (см ниже).

Другой проблемой может быть инверсия приоритетов, которая может проявиться при использовании семафоров процессами реального времени. Процессы реального времени могут быть прерваны операционной системой только для исполнения процессов с большим приоритетом. В этом случае процесс может заблокироваться по семафору в ожидании его отпускания

процессом с меньшим приоритетом. Если в это время будет работать процесс со средним между двумя процессами приоритетом, то процесс с высоким приоритетом может оказаться заблокированным на неограниченный промежуток времени[50].

Проблема инверсии приоритетов решается **наследованием приоритетов**[51]. По возможности семафоры могут быть заменены на мьютексы, поскольку у мьютексов наследование приоритетов может быть заранее предусмотрено. Таким образом, при захвате мьютекса потоком с большим приоритетом произойдёт упреждающее повышение приоритета у задачи, владеющей мьютексом, для его скорейшего отпускания[9].

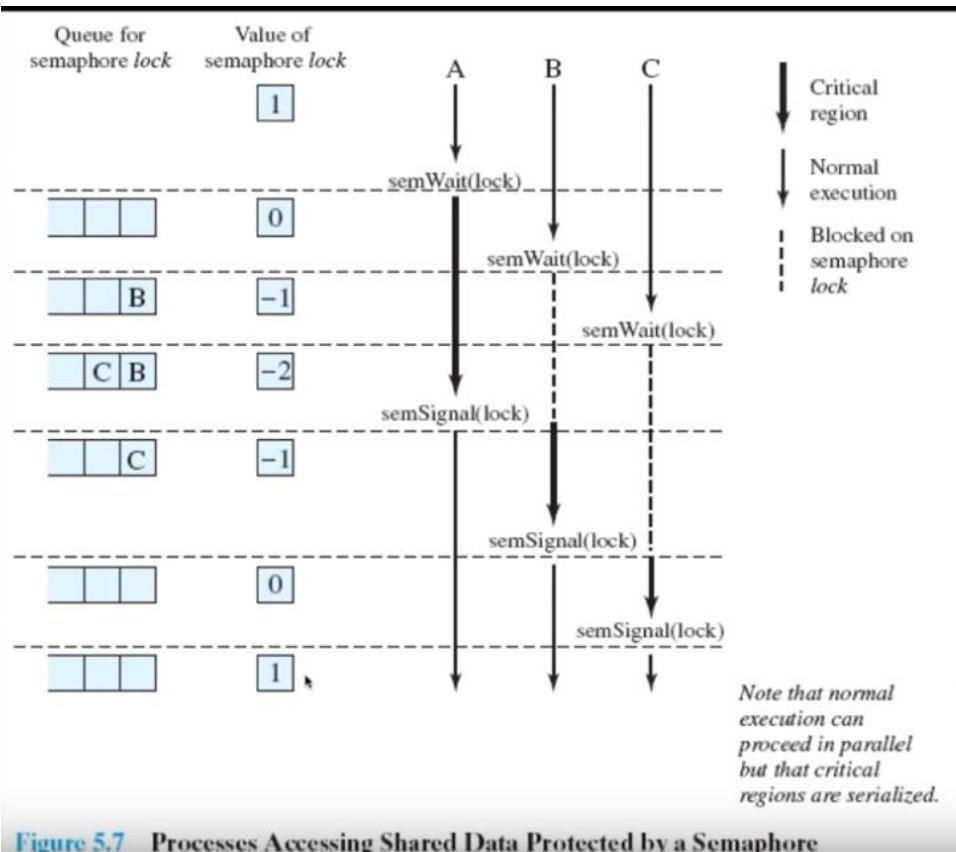


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

### 7.1.9 Разделяемая память

Является самым быстрым средством обмена данными между процессами[1].

В других средствах межпроцессового взаимодействия (IPC) обмен информацией между процессами проходит через ядро, что приводит к переключению контекста между процессом и ядром, т.е. к потерям производительности[2].

Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных

вызовов ядра. Сегмент разделяемой памяти подключается в свободную часть виртуального адресного пространства процесса[3]. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров.

#### Сценарий использования разделяемой памяти

- Сервер получает доступ к разделяемой памяти, используя семафор.
- Сервер производит запись данных в разделяемую память.
- После завершения записи данных сервер освобождает доступ к разделяемой памяти с помощью семафора.
- Клиент получает доступ к разделяемой памяти, запирая доступ к этой памяти для других процессов с помощью семафора.
- Клиент производит чтение данных из разделяемой памяти, а затем освобождает доступ к памяти с помощью семафора.

В программном обеспечении разделяемой памятью называют:

- Метод межпроцессного взаимодействия (IPC), то есть способ обмена данными между программами, работающими одновременно. Один процесс создаёт область в оперативной памяти, которая может быть доступна для других процессов.
- Метод экономии памяти, путём прямого обращения к тем исходным данным, которые при обычном подходе являются отдельными копиями исходных данных, вместо отображения виртуальной памяти или описанного метода . Такой подход обычно используется для разделяемых библиотек и для XIP.

Поскольку оба процесса могут получить доступ к общей области памяти как к обычной памяти, это очень быстрый способ связи (в отличие от других механизмов IPC, таких как именованные каналы, UNIX-сокеты или CORBA). С другой стороны, такой способ менее гибкий, например, обменивающиеся процессы должны быть запущены на одной машине (из перечисленных методов IPC только сетевые сокеты, не путать с сокетами домена UNIX, могут вести обмен данными через сеть), и необходимо быть внимательным, чтобы избежать

проблем при использовании разделяемой памяти на разных ядрах процессора и аппаратной архитектуре без когерентного (синхронизированного) кэша.

Обмен данными через разделяемую память используется, например, для передачи изображений между приложением и X-сервером на Unix системах.

Динамические библиотеки, как правило, загружаются в память один раз и отображены на несколько процессов, и только страницы, которые специфичны для отдельного процесса (поскольку отличаются некоторые идентификаторы) дублируются, как правило, с помощью механизма, известного как копирование-при-записи, который при попытке записи в разделяемую память незаметно для вызывающего запись процесса копирует страницы памяти, а затем записывает данные в эту копию.

В UNIX-подобных операционных системах POSIX предоставляет стандартизированное API для работы с разделяемой памятью — POSIX Shared Memory. Одной из ключевых особенностей операционных систем семейства UNIX является механизм копирования процессов (системный вызов `fork()`), который позволяет создавать анонимные участки разделяемой памяти перед копированием процесса и наследовать их процессами-потомками. После копирования процесса разделяемая память будет доступна как родительскому, так и дочернему процессу.

Некоторые библиотеки языка C++ предлагают доступ к работе с разделяемой памятью в кроссплатформенном виде. Например, библиотека Boost предоставляет класс `boost::interprocess::shared_memory_object` для POSIX-совместимых операционных систем, а библиотека Qt предоставляет класс `QSharedMemory`, унифицирующий доступ к разделяемой памяти для разных операционных систем с некоторыми ограничениями

### *7.1.10 Проецируемый в память файл mmap*

POSIX-совместимый системный вызов Unix, позволяющий выполнить отображение файла или устройства на память. Является методом ввода-вывода через отображение файла на память и естественным образом реализует выделение страниц по запросу, поскольку изначально содержимое файла не читается с диска и не использует физическую память вообще. Реальное считывание с диска производится в «ленивом» режиме, то есть при осуществлении доступа к определённому месту.

### *7.1.11 Очередь сообщений*

Очереди сообщений предоставляют асинхронный протокол передачи данных, означая, что отправитель и получатель сообщения не обязаны взаимодействовать с очередью сообщений одновременно. Размещённые в очереди сообщения хранятся до тех пор, пока получатель не получит их.

Очереди сообщений имеют неявные или явные ограничения на размер данных, которые могут передаваться в одном сообщении, и количество сообщений, которые могут оставаться в очереди.

Многие реализации очередей сообщений функционируют внутренне: внутри операционной системы или внутри приложения. Такие очереди существуют только для целей этой системы.

Другие реализации позволяют передавать сообщения между различными компьютерными системами, потенциально подключая несколько приложений и несколько операционных систем. Эти системы очередей сообщений обычно обеспечивают расширенную функциональность для обеспечения устойчивости, чтобы гарантировать, что сообщения не будут «потеряны» в случае сбоя системы.

### *7.1.12 Мэйлслот*

Мэйлслот является клиент-серверным интерфейсом. Сервер мэйлслотов — процесс, который создаёт мэйлслот и может читать из него информацию. Мэйлслот существует до тех пор, пока не закрыты все его серверные дескрипторы. Если несколько серверных процессов внутри домена создадут мэйлслоты с одинаковым именем, то сообщения, адресованные этому мэйлслоту и посылаемые в домен, будут приниматься всеми создавшими его процессами. Клиентом мэйлслота может быть любой процесс, знающий его имя. Клиент записывает в мэйлслот сообщения для передачи их посредством датаграмм серверу. Один и тот же процесс может быть одновременно клиентом и сервером мэйлслотов.

Для создания мэйлслотов используется специальное пространство имён «\\.\mailslot\[путь]имя». Для записи информации в мэйлслот на локальном компьютере клиентом используется то же самое имя, что использовалось сервером для создания мэйлслота.

### 7.1.13 Видеолекция

- Атомарная операция (atomic operation)
- Критическая секция (critical section)
- Взаимная блокировка (deadlock)
- Livelock
- Взаимное исключение (mutual exclusion)
- Состояние гонки (race condition)
- Resource Starvation

Resource starvation – ресурсное голодание

## Проблемы

- Распределение глобальных ресурсов
- Сложно обнаружить ошибки в программе так как результаты не всегда детерминистические и воспроизводимые

## Что нужно для взаимного исключения

- Только один процесс может находиться в критической секции для ресурса
- Процесс, который завершается в некритической секции не должен мешать другим процессам
- Нет взаимной блокировки и ресурсного голодания
- Процесс не должен ждать доступа к критической секции ресурса если он свободен
- Не должно быть никаких допущений о последовательности и количестве процессов
- Процесс может находиться в критической секции ограниченное количество времени

Критическая секция – туалет

## **7.2 Разделяемая память, средства синхронизации.**

См 7.1.9.

## **7.3 Очереди сообщений и другие средства обмена данными.**

См 7.1.11.

# **8**

## **8.1 Управление доступом к данным.**

Рассмотрим основные модели разграничения доступа к объектам компьютерных систем со стороны их субъектов. Понятие субъекта отличается от понятия пользователя компьютерной системы, которым обычно является физическое лицо, обладающее некоторой идентифицирующей его информацией (например, именем и паролем). Возможно существование в системе и псевдопользователей, например, системных процессов. Пользователь управляет работой субъекта (порожденного им из объекта-источника с программным кодом) с помощью его интерфейсных элементов (команд меню, кнопок и т. п.).

### *8.1.1 Избирательное (дискреционное) разграничение*

#### *8.1.1.1 Википедия (простое объяснение)*

Избирательное управление доступом (англ. discretionary access control, DAC) — управление доступом субъектов к объектам на основе списков управления доступом или матрицы доступа. Также используются названия дискреционное управление доступом, контролируемое управление доступом и разграничительное управление доступом.

Субъект доступа «Пользователь № 1» имеет право доступа только к объекту доступа № 3, поэтому его запрос к объекту доступа № 2 отклоняется. Субъект «Пользователь № 2» имеет право доступа как к объекту доступа № 1, так и к объекту доступа № 2, поэтому его запросы к данным объектам не отклоняются.

Для каждой пары (субъект — объект) должно быть задано явное и недвусмысленное перечисление допустимых типов доступа (читать, писать и т.

д.), то есть тех типов доступа, которые являются санкционированными для данного субъекта (индивидуа или группы индивидов) к данному ресурсу (объекту)

Возможны несколько подходов к построению дискреционного управления доступом:

- Каждый объект системы имеет привязанного к нему субъекта, называемого владельцем. Именно владелец устанавливает права доступа к объекту.
- Система имеет одного выделенного субъекта — суперпользователя, который имеет право устанавливать права владения для всех остальных субъектов системы.
- Субъект с определённым правом доступа может передать это право любому другому субъекту.

Возможны и смешанные варианты построения, когда одновременно в системе присутствуют как владельцы, устанавливающие права доступа к своим объектам, так и суперпользователь, имеющий возможность изменения прав для любого объекта и/или изменения его владельца. Именно такой смешанный вариант реализован в большинстве операционных систем, например Unix или Windows NT.

Избирательное управление доступом является основной реализацией разграничительной политики доступа к ресурсам при обработке конфиденциальных сведений, согласно требованиям к системе защиты информации.

**Системы с дискреционным контролем доступа разрешают пользователям полностью определять доступность их ресурсов, что означает, что они могут случайно или преднамеренно передать доступ неавторизованным пользователям.**

### 8.1.1.2 Другое объяснение

## Общая классификация субъектов и объектов доступа



Дискреционное разграничение доступа к объектам (Discretionary Access Control — DAC) характеризуется следующим набором свойств:

- все субъекты и объекты компьютерной системы должны быть однозначно идентифицированы;
- для любого объекта компьютерной системы определен пользователь-владелец;
- владелец объекта обладает правом определения прав доступа к объекту со стороны любых субъектов компьютерной системы;
- в компьютерной системе существует привилегированный пользователь, обладающий правом полного доступа к любому объекту (или правом становиться владельцем любого объекта).

Последнее свойство определяет невозможность существования в компьютерной системе потенциально недоступных объектов, владелец которых отсутствует. Но реализация права полного доступа к любому объекту посредством предварительного назначения себя его владельцем не позволяет привилегированному пользователю (администратору) использовать свои полномочия незаметно для реального владельца объекта.

Дискреционное разграничение доступа реализуется обычно в виде матрицы доступа, строки которой соответствуют субъектам компьютерной системы, а столбцы — ее объектам. Элементы матрицы доступа определяют права доступа субъектов к объектам. В целях сокращения затрат памяти матрица доступа может задаваться в виде списков прав субъектов (для каждого из них

создается список всех объектов, к которым разрешен доступ со стороны данного субъекта) или в виде списков контроля доступа (для каждого объекта информационной системы создается список всех субъектов, которым разрешен доступ к данному объекту).

К достоинствам дискреционного разграничения доступа относятся относительно простая реализация (проверка прав доступа субъекта к объекту производится в момент открытия этого объекта в процессе субъекта) и хорошая изученность (в наиболее распространенных операционных системах универсального назначения типа Microsoft Windows и Unix применяется именно эта модель разграничения доступа).

Остановимся на недостатках дискреционного разграничения доступа. Прежде всего, к ним относится статичность разграничения доступа — права доступа к уже открытому субъектом объекту в дальнейшем не изменяются независимо от изменения состояния компьютерной системы.

При использовании дискреционного разграничения доступа не существует возможности проверки, не приведет ли разрешение доступа к объекту для некоторого субъекта к нарушению безопасности информации в компьютерной системе (например, владелец базы данных с конфиденциальной информацией, дав разрешение на ее чтение другому пользователю, делает этого пользователя фактически владельцем защищаемой информации). Иначе говоря, дискреционное разграничение доступа не обеспечивает защиты от утечки конфиденциальной информации.

Наконец, к недостаткам дискреционного управления доступом относится автоматическое назначение прав доступа субъектам (из-за большого количества объектов в информационной системе в качестве субъектов доступа остаются только ее пользователи, а значение элемента матрицы доступа вычисляется с помощью функции, определяющей права доступа порожденного пользователем субъекта к данному объекту компьютерной системы).

### *8.1.2 Мандатное разграничение*

Mandatory Access Control — MAC.

Самое важное достоинство заключается в том, что пользователь не может полностью управлять доступом к ресурсам, которые он создаёт.

К основным характеристикам этой модели относится следующее:

- все субъекты и объекты компьютерной системы должны быть однозначно идентифицированы;

- имеется линейно упорядоченный набор меток конфиденциальности и соответствующих им уровней (степеней) допуска (нулевая метка или степень соответствуют общедоступному объекту и степени допуска к работе только с общедоступными объектами);
- каждому объекту компьютерной системы присвоена метка конфиденциальности;
- каждому субъекту компьютерной системы присваивается степень допуска;
- в процессе своего существования каждый субъект имеет свой уровень конфиденциальности, равный максимуму из меток конфиденциальности объектов, к которым данный субъект получил доступ;
- в компьютерной системе существует привилегированный пользователь, имеющий полномочия на удаление любого объекта системы;
- понизить метку конфиденциальности объекта может только субъект, имеющий доступ к данному объекту и обладающий специальной привилегией;
- право на чтение информации из объекта получает только тот субъект, чья степень допуска не меньше метки конфиденциальности данного объекта (правило «не читать выше»);
- право на запись информации в объект получает только тот субъект, чей уровень конфиденциальности не больше метки конфиденциальности данного объекта (правило «не записывать ниже»).

Основной целью мандатного разграничения доступа к объектам является предотвращение утечки информации из объектов с высокой меткой конфиденциальности в объекты с низкой меткой конфиденциальности (противодействие созданию каналов передачи информации «сверху вниз»).

Для мандатного разграничения доступа к объектам компьютерной системы формально доказано следующее важное утверждение: если начальное состояние компьютерной системы безопасно и все переходы из одного состояния системы в другое не нарушают правил разграничения доступа, то любое последующее состояние компьютерной системы также безопасно.

К другим достоинствам мандатного разграничения доступа относятся:

- более высокая надежность работы самой компьютерной системы, так как при разграничении доступа к объектам контролируется и состояние самой системы, а не только соблюдение установленных правил;
- большая простота определения правил разграничения доступа по сравнению с дискреционным разграничением (эти правила более ясны для разработчиков и пользователей компьютерной системы).

Отметим недостатки мандатного разграничения доступа к объектам компьютерной системы:

- сложность программной реализации, что увеличивает вероятность внесения ошибок и появления каналов утечки конфиденциальной информации;
- снижение эффективности работы компьютерной системы, так как проверка прав доступа субъекта к объекту выполняется **не только при открытии объекта в процессе субъекта, но и перед выполнением любой операции чтения из объекта или записи в объект**;
- создание дополнительных неудобств работе пользователей компьютерной системы, связанных с невозможностью изменения информации в неконфиденциальном объекте, если тот же самый процесс использует информацию из конфиденциального объекта (его уровень конфиденциальности больше нуля).

Преодоление последнего недостатка требует разработки программного обеспечения компьютерной системы с учетом особенностей мандатного разграничения доступа.

Из-за отмеченных недостатков мандатного разграничения доступа в реальных компьютерных системах множество объектов, к которым применяется мандатное разграничение, является подмножеством объектов, доступ к которым осуществляется на основе дискреционного разграничения.

При использовании мандатного разграничения доступа к объектам необходимо также обеспечить достоверное подтверждение назначенного пользователю уровня допуска даже при отсутствии защищенного канала связи с сервером аутентификации. Это может быть обеспечено при использовании инфраструктуры открытых ключей.

Рассмотрим преимущества подобного способа хранения информации об уровне допуска пользователя к ресурсам компьютерной системы перед ее хранением в локальной или глобальной (доменной) учетной записи пользователя. При работе в сети сертификат пользователя (при его отсутствии в

локальном хранилище сертификатов) может быть всегда запрошен в корпоративном удостоверяющем центре. При попытке получения доступа к ресурсам автономного (не подключенного к сети) компьютера сертификат может быть импортирован из файла на предоставленном пользователем носителе (флэш-диске, дискете, компакт-диске) или непосредственно с устройства аутентификации пользователя (смарт-карты, USB-ключа).

При импорте сертификата из файла пользователю потребуется ввести пароль для генерации сеансового ключа расшифрования личного (закрытого) криптографического ключа, а при импорте с устройства аутентификации — PIN-код. Это обеспечит защиту от попытки использования похищенного носителя.

В Руководящем документе России «Автоматизированные системы. Защита от несанкционированного доступа к информации. Классификация автоматизированных систем и требования по защите информации» содержатся следующие требования по разграничению доступа субъектов к защищенным объектам автоматизированных систем:

- должен осуществляться контроль доступа субъектов к защищаемым ресурсам в соответствии с матрицей доступа (дискреционное разграничение доступа — классы защищенности 1Г, 1В, 1Б, 1А);
- должно осуществляться управление потоками информации с помощью меток конфиденциальности (мандатное разграничение доступа). При этом уровень конфиденциальности накопителей должен быть не ниже уровня конфиденциальности записываемой на них информации (классы защищенности 1В, 1Б, 1А).

### *8.1.3 Ролевое разграничение*

Ролевое разграничение доступа (Role-Based Access Control — RBAC) основано на том соображении, что в реальной жизни организации ее сотрудники выполняют определенные функциональные обязанности не от своего имени, а в рамках некоторой занимаемой ими должности (или роли). Реализация ролевого разграничения доступа к объектам компьютерной системы требует разработки набора (библиотеки) ролей, определяемых как набор прав доступа к объектам информационной системы (прав на выполнение над ними определенного набора действий). Этот набор прав должен соответствовать выполняемой работником трудовой функции.

Такое разграничение доступа является составляющей многих современных компьютерных систем. Как правило, данный подход применяется в системах

защиты СУБД, а отдельные элементы реализуются в сетевых операционных системах. Ролевой подход часто используется в системах, для пользователей которых чётко определён круг их должностных полномочий и обязанностей.

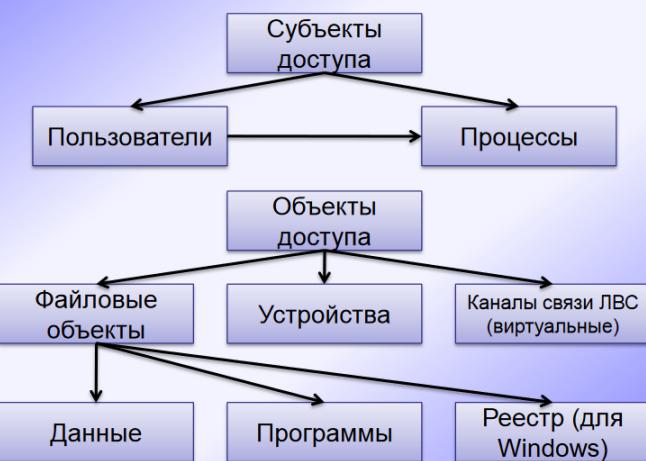
Несмотря на то, что Роль является совокупностью прав доступа на объекты компьютерной системы, ролевое управление доступом отнюдь не является частным случаем избирательного управления доступом, так как его правила определяют порядок предоставления доступа субъектам компьютерной системы в зависимости от имеющихся (или отсутствующих) у него ролей в каждый момент времени, что является характерным для систем мандатного управления доступом. С другой стороны, правила ролевого разграничения доступа являются более гибкими, чем при мандатном подходе к разграничению.

Так как привилегии не назначаются пользователям непосредственно и приобретаются ими только через свою роль (или роли), управление индивидуальными правами пользователя по сути сводится к назначению ему ролей. Это упрощает такие операции, управление множеством пользователей.

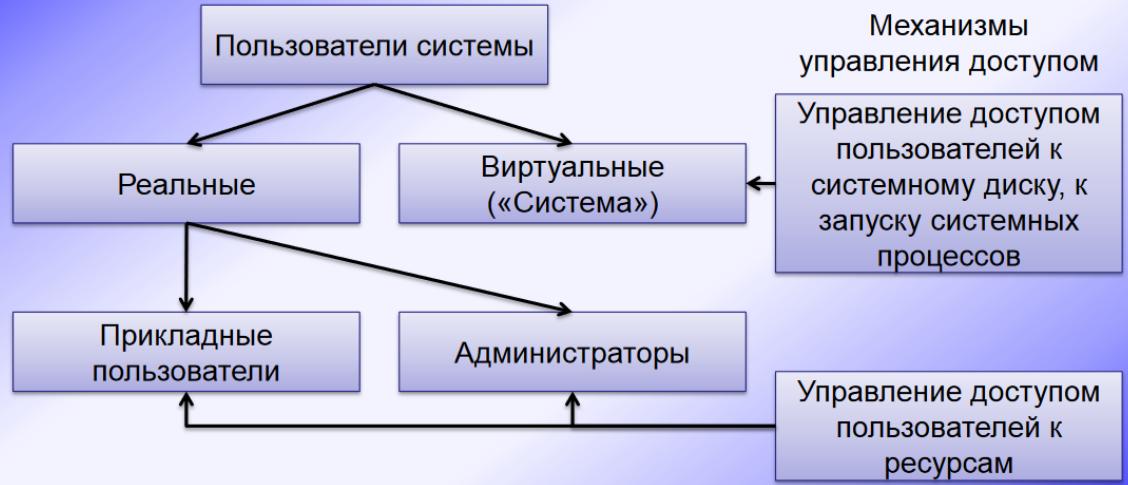
#### 8.1.4 Презентация

<https://www.prorobot.ru/referats/r04/prorobot.ru-04-0031.pdf>

### Общая классификация субъектов и объектов доступа



# Классификация пользователей



# Классификация процессов

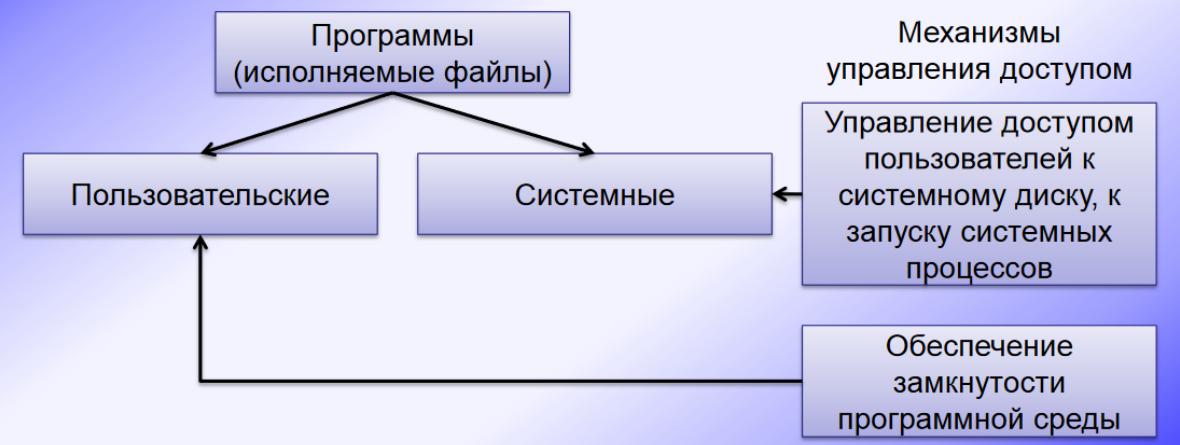


# Классификация файловых объектов данных



6

# Классификация файловых объектов программ



## 8.2 Файловые системы

[https://portal.tpu.ru/SHARED/k/KATSMAN/methodwork/Tab2/Textbook\\_Zamyatkin.pdf](https://portal.tpu.ru/SHARED/k/KATSMAN/methodwork/Tab2/Textbook_Zamyatkin.pdf)

*Файловая система* – это часть ОС, организующая работу с данными, хранящимися во внешней памяти, и обеспечивающая пользователю удобный интерфейс при работе с такими данными. Термин *файловая система* определяет, прежде всего, принципы доступа к данным, организованным в файлы. Говоря о файловых системах, иногда употребляют термин *система управления файлами*, под которой следует понимать некую конкретную реализацию файловой системы, то есть комплекс программных модулей, обеспечивающих работу с файлами в конкретной ОС.

*Файл* – это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Файлы хранятся в энергонезависимой памяти, обычно – на магнитных дисках. Основными целями использования файла являются:

1. Долговременное и надежное хранение информации. Долговременность достигается за счет использования запоминающих устройств, не зависящих от питания, а высокая надежность определяется средствами защиты доступа к файлам и общей организацией программного кода ОС, при которой сбои аппаратуры чаще всего не разрушают информацию, хранящуюся в файлах.

2. Совместное использование информации. Файлы обеспечивают естественный и легкий способ разделения информации между приложениями и пользователями за счет наличия понятного человеку символического имени и постоянства хранимой информации и расположения файла. Пользователь должен иметь удобные средства работы с файлами, включая каталоги-справочники, объединяющие файлы в группы, средства поиска файлов по признакам, набор команд для создания, модификации и удаления файлов. Файл может быть создан одним пользователем, а затем использоваться другим пользователем, при этом создатель файла или администратор могут определить права доступа к нему других пользователей.

Файловая система, являющаяся неотъемлемой частью любой современной ОС, включает:

- совокупность всех файлов на диске;
- наборы структур данных, используемых для управления файлами (каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске);
- комплекс системных программных средств, реализующих различные операции над файлами (создание, уничтожение, чтение, запись, именование и поиск файлов).

Файловая система позволяет программам обходиться набором относительно простых операций для выполнения действий над некоторым аб-

структурным объектом, представляющим файл. При этом программистам не нужно иметь дело с деталями действительного расположения данных на диске, буферизацией данных и другими низкоуровневыми проблемами передачи данных с долговременного запоминающего устройства – все эти функции файловая система берет на себя. Файловая система распределяет дисковую память, поддерживает именование файлов, отображает имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление файлов.

### 8.2.1 Типы файлов

Файловые системы поддерживают несколько функционально различных типов файлов, в число которых, как правило, входят:

- обычные файлы;
- файлы-каталоги;
- специальные файлы;
- отображаемые в память файлы;
- именованные конвейеры;

*Файлы-каталоги* или просто *каталоги* – это особый тип файлов, которые содержат системную справочную информацию о наборе файлов, сгруппированных пользователями по какому-либо признаку (например, в одну группу объединяются файлы, содержащие документы одного договора, или файлы, составляющие один программный пакет), с другой стороны – это файл, содержащий системную информацию о группе файлов, его составляющих.

*Специальные файлы* – это фиктивные файлы, ассоциированные с устройствами ввода-вывода, которые используются для унификации механизма доступа к файлам и внешним устройствам. Специальные файлы позволяют пользователю выполнять операции ввода-вывода посредством обычных команд записи в файл или чтения из файла. Эти команды обрабатываются сначала программами файловой системы, а затем на некотором этапе выполнения запроса преобразуются ОС в команды управления соответствующим устройством.

*Отображаемые в память файлы* (англ. *memory-mapped files*) – это мощная возможность ОС, позволяющая приложениям осуществлять доступ к файлам на диске тем же самым способом, каким осуществляется доступ к динамической памяти, то есть через указатели. Смысл отображения файла в память заключается в том, что содержимое файла (или часть содержимого) отображается в некоторый диапазон виртуального адресного пространства процесса, после чего обращение по какому-либо адресу из этого диапазона означает обращение к файлу на диске. Естественно, не каждое обращение к отображеному в память файлу вызывает операцию чтения/записи. Менеджер виртуальной памяти кэширует обращения к диску и тем самым обеспечивает высокую эффективность работы с отображенными файлами.

*Именованные конвейеры* (именованные каналы) – одно из средств межпроцессного взаимодействия, детали которого рассмотрены в п. 3.3.6.

~~к диску~~. Иногда прибегают к модификации подхода связного списка, организуя хранение адресов  $n$  свободных блоков в первом свободном блоке: первые  $n - 1$  этих блоков используются для хранения данных, а последний – содержит адреса других  $n$  блоков, и т. д.

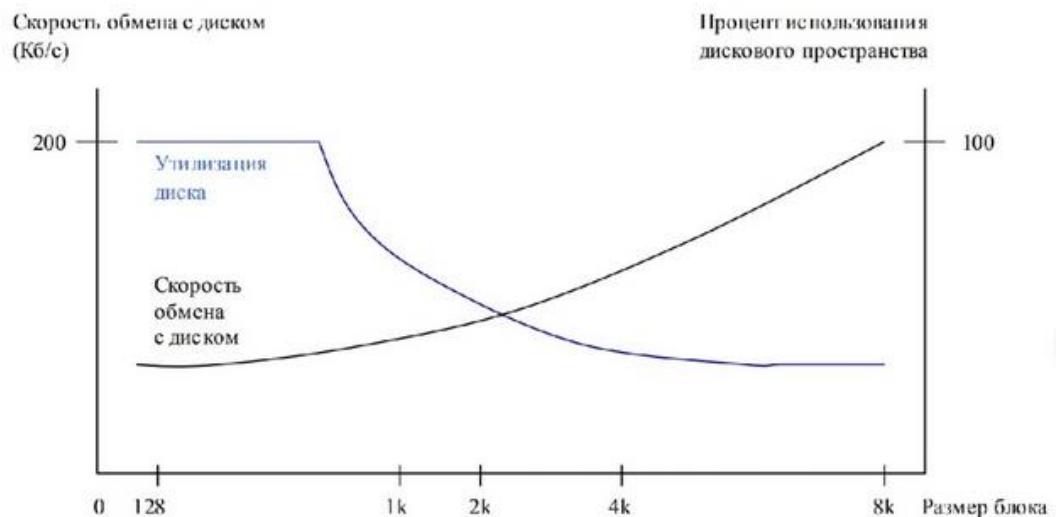


Рис. 42. Результаты исследований при определении оптимального размера блока

## *8.2.2 Атрибуты файлов*

Кроме имени ОС часто связывают с каждым файлом и другую информацию, например дату модификации, размер и т. д. Эти характеристики файлов называют *атрибутами*. В разных файловых системах могут использоваться в качестве атрибутов разные характеристики. Например, такими характеристиками могут быть следующие:

- информация о разрешенном доступе;
- пароль для доступа к файлу;
- владелец файла;
- создатель файла;
- признак «только для чтения»;
- признак «скрытый файл»;
- признак «системный файл»;
- признак «архивный файл»;
- признак «двоичный/символьный»;
- признак «временный» (удалить после завершения процесса);
- признак блокировки;
- длина записи;
- указатель на ключевое поле в записи;
- длина ключа;
- время создания, последнего доступа и последнего изменения;
- текущий размер файла;
- максимальный размер файла.

## *8.2.3 Особенности файловых систем*

### *– FAT*

Еще один недостаток *FAT* заключается в том, что ее производительность сильно зависит от количества файлов, хранящихся в одном каталоге. При большом количестве файлов (~1000), выполнение операции считывания списка файлов в каталоге может занять несколько минут. Это обусловлено тем, что в *FAT* каталог имеет линейную неупорядоченную структуру, и имена файлов в каталогах идут в порядке их создания. В результате чем больше в каталоге записей, тем медленнее работают программы, так как при поиске файла требуется просмотреть последовательно все записи в каталоге.

*NTFS*

*ext234*

От файловой системы не требуется, чтобы она целиком размещалась на том устройстве, где находится корень. Запрос от системы *mount* (на установку носителей и т. п.) позволяет встраивать в иерархию файлов файлы на сменных томах. Команда *mount* имеет несколько аргументов, но обязательных аргументов у стандартного варианта ее использования два: имя файла блочного устройства и имя каталога. В результате выполнения этой команды файловая подсистема, расположенная на указанном устройстве, подключается к системе таким образом, что ее содержимое заменяет собой содержимое заданного в команде каталога. Поэтому для монтирования соответствующего тома обычно используют пустой каталог. Команда *umount* выполняет обратную операцию – «отсоединяет» файловую систему, после чего диск с данными можно физически извлечь из системы. Например, для записи данных на дискету необходимо ее «подмонтировать», а после работы – «размонтировать».

Монтирование файловых систем позволяет получить единое логическое файловое пространство, в то время как реально отдельные каталоги с файлами могут находиться в разных разделах одного жесткого диска и даже на разных жестких дисках. Причем, как отмечено выше, сами файловые системы для монтируемых разделов могут быть разными. Например, при работе в ОС *Linux* можно иметь часть разделов с файловой системой *EXT2FS*, а часть разделов – с файловой системой *EXT3FS*.

#### 8.2.4 Реализация ФС

Ключевой вопрос реализации файловой системы – способ связывания файлов с блоками диска. В ОС используется несколько способов выделения файлу дискового пространства, для каждого из которых сведения о локализации блоков данных файла можно извлечь из записи в директории, соответствующей символному имени файла.

##### 8.2.4.1 Выделение непрерывной последовательностью блоков

**Выделение непрерывной последовательностью блоков.** Простейший способ – хранить каждый файл как непрерывную последова-

тельность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока  $b$ , занимает затем блоки  $b + 1, b + 2, \dots, b + n - 1$ . Этот способ имеет два преимущества:

- легкая реализация, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок;
- обеспечивает хорошую производительность, потому что целый файл может быть считан за одну дисковую операцию.

Например, непрерывное выделение использовано в ОС *IBM/CMS, RSX-11* (для исполняемых файлов).

Основная проблема, в связи с которой этот способ мало распространен, – трудности в поиске места для нового файла. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения  $n$  блоков из списка свободных фрагментов. Наиболее распространенные стратегии решения этой проблемы – выбор первого подходящего по размеру блока (англ. *first fit*), наиболее подходящего, т. е. того, при размещении в котором наиболее тесно (англ. *best fit*), и наименее подходящего, т. е. выбирается наибольший блок (англ. *worst fit*).

Способ характеризуется наличием существенной *внешней фрагментации* (в большей или меньшей степени – в зависимости от размера диска и среднего размера файла). Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока не известен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании), но чаще – трудно. Если места недостаточно, то пользовательская программа может быть приостановлена, предполагая выделение дополнительного места для файла при последующем перезапуске. Некоторые ОС используют модифицированный вариант непрерывного выделения: «основные блоки файла» + «резервные блоки». Однако с выделением блоков из резерва возникают те же проблемы, так как возникает задача выделения непрерывной последовательности блоков диска теперь уже из совокупности резервных блоков.

#### 8.2.4.2 Выделение связным списком

**Выделение связным списком.** Метод распределения блоков в виде связного списка решает основную проблему непрерывного выделения, то есть устраняет внешнюю фрагментацию (рис. 39). Каждый файл – связный список блоков диска. Запись в директории содержит указатель на первый и последний блоки файла. Каждый блок содержит указатель на следующий блок.

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса.

Отметим, что нет необходимости декларировать размер файла в момент создания, и поэтому файл может неограниченно расти. Связное выделение имеет несколько существенных недостатков:

- при прямом доступе к файлу для поиска  $i$ -го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до  $i - 1$ , т. е. выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени;
- прямым следствием этого является низкая *надежность* – наличие дефектного блока в списке приводит к потере информации в остаточной части файла и потенциально к потере дискового пространства, отведенного под этот файл;
- для указателя на следующий блок внутри блока нужно выделить место размером, традиционно определяемым степенью двойки (многие программы читают и записывают блоками по степеням двойки), который перестает быть таковым, так как указатель отбирает несколько байтов.

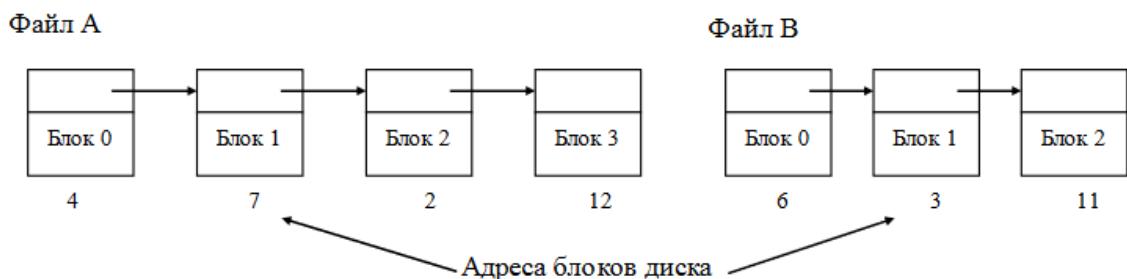


Рис. 39. Хранение файлов в связных списках дисковых блоков

В связи с вышеизложенным метод связного списка обычно не используется в чистом виде.

#### 8.2.4.3 Распределение связным списком с использованием индекса

##### **Распределение связным списком с использованием индекса.**

Недостатки предыдущего способа могут быть устранины путем изъятия указателя из каждого дискового блока и помещения его в индексную таблицу в памяти, которая называется *таблицей размещения файлов* (англ. *file allocation table – FAT*). Этой схемы придерживаются многие ОС (*MS-DOS*, *OS/2*, *MS Windows* и др.).

По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы *FAT* можно локализовать блоки файла независимо от его размера (значение равно адресу следующего блока этого файла). В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например *EOF* (рис. 40).

Номер блоков диска		
1	0	
2	10	
3	11	Начало файла F2
4	0	
5	EOF	
6	2	Начало файла F1
7	EOF	
8	0	
9	0	
10	7	
11	5	

Рис. 40. Способ выделения памяти с использованием связного списка в ОП

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы. Более подробно особенности использования таблицы размещения файлов рассмотрены в п. 5.5.1 при описании файловой системы FAT.

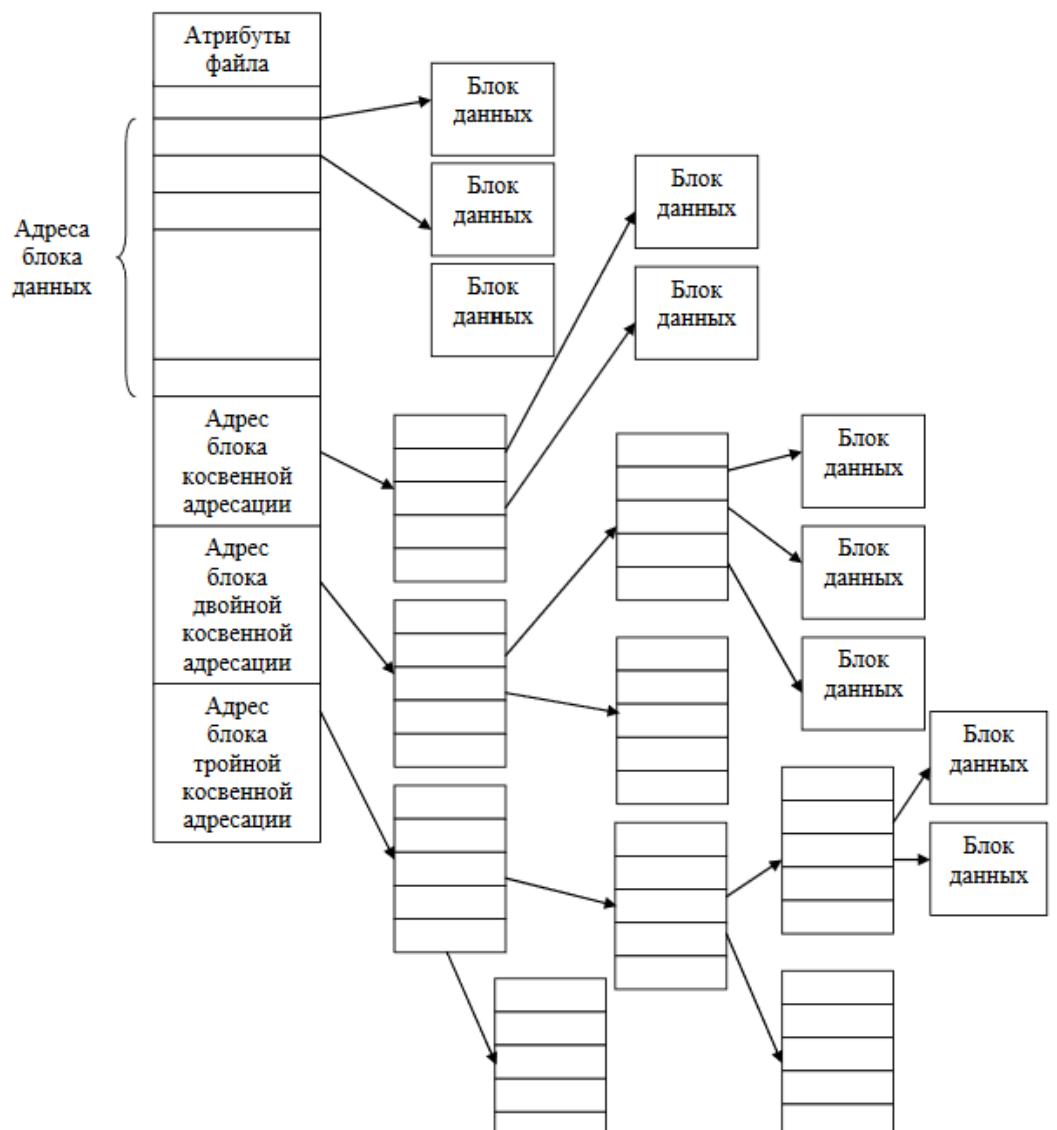
#### 8.2.4.4 Индексные узлы

**Индексные узлы.** Четвертый и последний способ «выяснения принадлежности» блока к файлу – связать с каждым файлом маленькую таблицу, называемую *индексным узлом* (*i-node*), которая перечисляет атрибуты и дисковые адреса блоков файла (рис. 41). Каждый файл имеет свой собственный индексный блок, который содержит адреса блоков данных. Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения. Индексирование поддерживает прямой доступ к файлу без ущерба от внешней фрагментации.

Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, поэтому для маленьких файлов индексный узел хранит всю необходимую информацию, которая копируется с диска в память в момент открытия файла. Для больших файлов один из адресов индексного узла указывает на блок *косвенной адресации*. Этот блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок *двойной косвенной адресации*, который содержит адреса блоков косвенной адресации. Если и этого недостаточно, используют блок *тройной косвенной адресации*.

Эту схему распределения внешней памяти использует ОС *Unix*, а также файловые системы *HPFS*, *NTFS* и др. Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от не-

скольких байт до нескольких гигабайт. Существенно то, что для леньких файлов используется только прямая адресация, обеспечивающая высокую производительность.



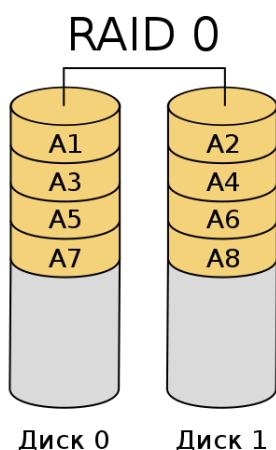
## 8.2.5 RAID

Дисковый массив *RAID*<sup>22</sup> – это консолидированная серверная система для хранения данных большого объема, в которой для размещения информации используют несколько жестких дисков. Поддержка различных уровней избыточности, производительности и способов восстановления после сбоя осуществляется посредством целого ряда разнообразных методик хранения. В массивах *RAID* значительное число дисков относительно малой емкости используется для хранения крупных объемов данных, а также для обеспечения более высокой надежности и избыточности.

Дисковый массив *RAID* может быть образован несколькими способами. Некоторые типы массивов *RAID* предназначены в первую очередь для повышения производительности, гарантии высокого уровня надежности, обеспечения отказоустойчивости и коррекции ошибок. Общей функцией для массивов *RAID* является функция «горячей замены». Иными словами, пользователь имеет возможность удалить выбранный дисковод, установив на его место другой. Для большинства типов дисковых массивов *RAID* данные на замененном диске могут быть восстановлены автоматически, без отключения сервера.

### 8.2.5.1 Raid 0

- RAID 0** (*striping* — «чередование») — дисковый массив из двух или более жёстких дисков без резервирования. Информация разбивается на блоки данных ( $A_i$ ) фиксированной длины и записывается на оба/несколько дисков поочередно, то есть один блок(A1) на первый диск, а второй блок(A2) на второй диск соответственно.
- (+): Скорость считывания файлов увеличивается в  $n$  раз, где  $n$  — количество дисков. При этом такая оптимальная производительность достигается только для больших запросов, когда фрагменты файла находятся на каждом из дисков.
- (-): Увеличивается вероятность потери данных: если вероятность отказа 1 диска равна  $p$ , то вероятность выхода из строя массива RAID 0 из двух дисков равна  $2p-p^2$ . Таким образом, если вероятность отказа одного диска за год равна 1 %, то вероятность отказа массива RAID0 из двух дисков составляет 1,99 %, то есть практически в два раза больше.



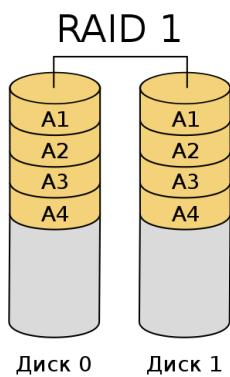
### 8.2.5.2 Raid 1

**RAID 1** (*mirroring — «зеркалирование»*) — массив из двух (или более) дисков, являющихся полными копиями друг друга. Не следует путать с массивами RAID 1+0 (RAID 10), RAID 0+1 (RAID 01), в которых используются более сложные механизмы зеркалирования.

(+): Обеспечивает приемлемую скорость записи (такую же, как и без дублирования) и выигрыш по скорости чтения при распараллеливании запросов<sup>[2]</sup>.

(+): Имеет высокую надёжность — работает до тех пор, пока функционирует хотя бы один диск в массиве. Вероятность выхода из строя сразу двух дисков равна произведению вероятностей отказа каждого диска, то есть значительно ниже вероятности выхода из строя отдельного диска. На практике при выходе из строя одного из дисков следует срочно принимать меры — вновь восстанавливать избыточность. Для этого с любым уровнем RAID (кроме нулевого) рекомендуют использовать диски горячего резерва.

(-): Недостаток RAID 1 в том, что по цене двух (и более) жестких дисков пользователь фактически получает объём лишь одного.



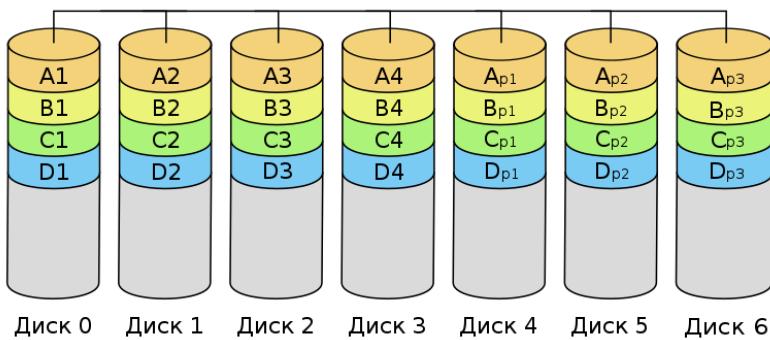
### 8.2.5.3 Raid 2

Массивы такого типа основаны на использовании кода Хэмминга. Диски делятся на две группы: для данных и для кодов коррекции ошибок, причём если данные хранятся на  $2^n - n - 1$  дисках, то для хранения кодов коррекции необходимо  $n$  дисков. Суммарное количество дисков при этом будет равняться  $2^n - 1$ . Данные распределяются по дискам, предназначенным для хранения информации, так же, как и в RAID 0, то есть они разбиваются на небольшие блоки по числу дисков. Оставшиеся диски хранят коды коррекции ошибок, по которым в случае выхода какого-либо жёсткого диска из строя возможно восстановление информации. Метод Хэмминга давно применяется в памяти типа **ECC** и позволяет на лету исправлять однократные и обнаруживать двукратные ошибки.

**Достоинством** массива RAID 2 является повышение скорости дисковых операций по сравнению с производительностью одного диска.

**Недостатком** массива RAID 2 является то, что минимальное количество дисков, при котором имеет смысл его использовать — 7, только начиная с этого количества для него требуется меньше дисков, чем для RAID 1 (4 диска с данными, 3 диска с кодами коррекции ошибок), в дальнейшем избыточность уменьшается по экспоненте.

## RAID 2



### 8.2.5.4 Raid 3

В массиве RAID 3 из  $n$  дисков данные разбиваются на куски размером меньше сектора (разбиваются на байты или блоки) и распределяются по  $n - 1$  дискам. Ещё один диск используется для хранения блоков чётности. В RAID 2 для этой цели применялся  $n - 1$  диск, но большая часть информации на контрольных дисках использовалась для коррекции ошибок «на лету», в то же время большинство пользователей устраивает простое восстановление информации в случае её повреждения, для чего хватает данных, умещающихся на одном выделенном жёстком диске.

Отличия RAID 3 от RAID 2: невозможность коррекции ошибок на лету.

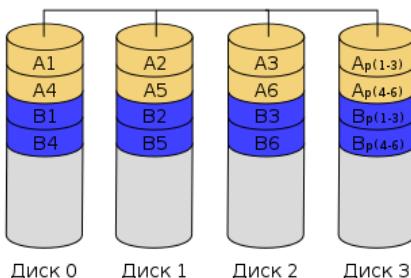
#### Достоинства:

- высокая скорость чтения и записи данных;
- минимальное количество дисков для создания массива равно трём.

#### Недостатки:

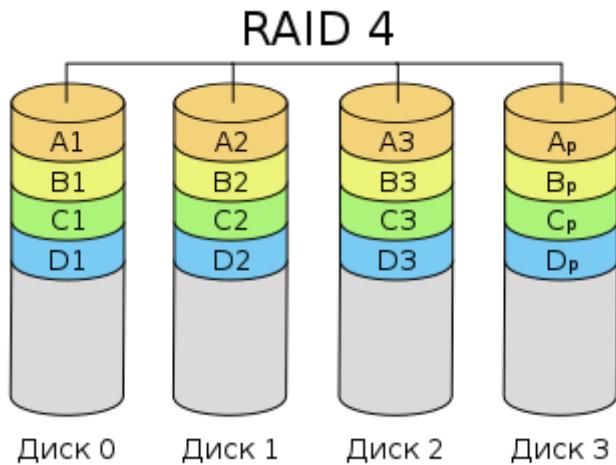
- массив этого типа хорош только для однозадачной работы с большими файлами, так как время доступа к отдельному сектору, разбитому по дискам, равно максимальному из интервалов доступа к секторам каждого из дисков. Для блоков малого размера время доступа намного больше времени чтения.
- большая нагрузка на контрольный диск, и, как следствие, его надёжность сильно падает по сравнению с дисками, хранящими данные.

## RAID 3



### 8.2.5.5 Raid 4

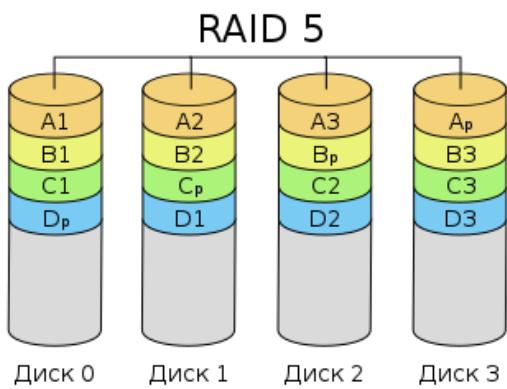
RAID 4 похож на RAID 3, но отличается от него тем, что данные разбиваются на блоки, а не на байты. Таким образом, удалось отчасти «победить» проблему низкой скорости передачи данных небольшого объёма. Запись же производится медленно из-за того, что чётность для блока генерируется при записи и записывается на единственный диск.



### 8.2.5.6 Raid 5

Основным недостатком уровней RAID от 2-го до 4-го является невозможность производить параллельные операции записи, так как для хранения информации о чётности используется отдельный контрольный диск. RAID 5 не имеет этого недостатка. Блоки данных и контрольные суммы циклически записываются на все диски массива, нет асимметричности конфигурации дисков. Под контрольными суммами подразумевается результат операции XOR (исключающее или). XOR обладает особенностью, которая даёт возможность заменить любой operand результатом, и, применив алгоритм XOR, получить в результате недостающий operand. Например:  $a \text{ xor } b = c$  (где  $a, b, c$  — три диска RAID-массива), в случае если  $a$  откажет, мы можем получить его, поставив на его место  $c$  и проведя XOR между  $c$  и  $b$ :  $c \text{ xor } b = a$ . Это применимо вне зависимости от количества operandов:  $a \text{ xor } b \text{ xor } c \text{ xor } d = e$ . Если отказывает  $c$ , тогда  $e$  встаёт на его место и, проведя XOR, в результате получаем  $c$ :  $a \text{ xor } b \text{ xor } e \text{ xor } d = c$ . Этот метод по сути обеспечивает отказоустойчивость 5 версии. Для хранения результата XOR требуется всего 1 диск, размер которого равен размеру любого другого диска в RAID.

Минимальное количество используемых дисков равно трём.



## Достоинства

RAID 5 получил широкое распространение, в первую очередь благодаря своей экономичности. Объём дискового массива RAID 5 рассчитывается по формуле  $(n-1) * \text{hddsize}$ , где  $n$  — число дисков в массиве, а  $\text{hddsize}$  — размер диска (наименьшего, если диски имеют разный размер). Например, для массива из четырёх дисков по 80 гигабайт общий объём будет  $(4 - 1) * 80 = 240$  гигабайт, то есть «потеряется» всего 25 % против 50 % RAID 10. И с увеличением количества дисков в массиве экономия (по сравнению с другими уровнями RAID, обладающими отказоустойчивостью) продолжает увеличиваться.

RAID 5 обеспечивает высокую скорость чтения — выигрыш достигается за счёт независимых потоков данных с нескольких дисков массива, которые могут обрабатываться параллельно.

## Недостатки

Производительность RAID 5 заметно ниже на операциях типа Random Write (записи в произвольном порядке), при которых производительность падает на 10-25 % от производительности RAID 0 (или RAID 10), так как требует большего количества операций с дисками (каждая операция записи, за исключением так называемых full-stripe write-ов, заменяется на контроллере RAID на четыре — две операции чтения и две операции записи).

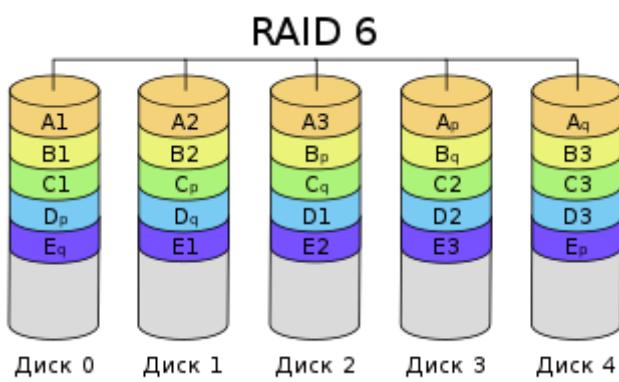
При выходе из строя одного диска надёжность тома сразу снижается до уровня RAID 0 с соответствующим количеством дисков  $n-1$  — то есть в  $n-1$  раз ниже надёжности одного диска — данное состояние называется критическим (degrade или critical). Для возвращения массива к нормальной работе требуется длительный процесс восстановления, связанный с ощутимой потерей производительности и повышенным риском.

В ходе восстановления (rebuild или reconstruction) контроллер осуществляет длительное интенсивное чтение, которое может спровоцировать выход из строя ещё одного или нескольких дисков массива. Кроме того, в ходе

чтения могут выявляться ранее не обнаруженные сбои чтения в массивах cold data (данных, к которым не обращаются при обычной работе массива, архивные и малоактивные данные), препятствующие восстановлению. Если до полного восстановления массива произойдет выход из строя, или возникнет невосстановимая ошибка чтения хотя бы на ещё одном диске, то массив разрушается и данные на нём восстановлению обычными методами не подлежат. Для предотвращения таких ситуаций в RAID-контроллерах может применяться анализ атрибутов S.M.A.R.T.

#### 8.2.5.7 Raid 6

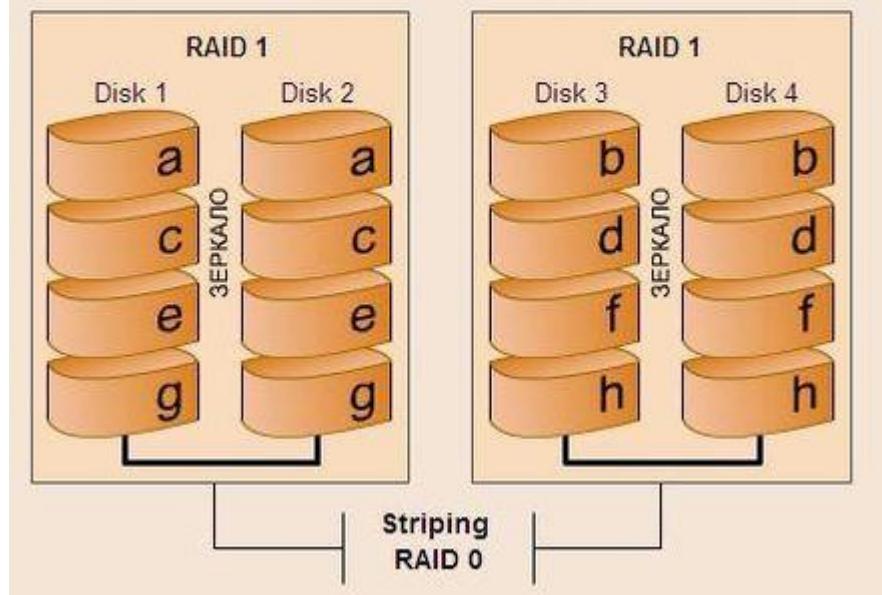
RAID 6 — похож на RAID 5, но имеет более высокую степень надёжности — два (или более) диска данных и два диска контроля чётности. Основан на кодах Рида — Соломона и обеспечивает работоспособность после одновременного выхода из строя любых двух дисков. Обычно использование RAID 6 вызывает примерно 10-15 % падение производительности дисковой группы, относительно RAID 5, что вызвано большим объёмом работы для контроллера (более сложный алгоритм расчёта контрольных сумм), а также необходимостью читать и перезаписывать больше дисковых блоков при записи каждого блока.



### 8.2.5.8 Raid 10

#### RAID 10

Отказоустойчивый массив с дублированием и параллельной обработкой



### 8.2.6 Вики

Файловая система (англ. file system) — порядок, определяющий способ организации, хранения и именования данных на носителях информации в компьютерах, а также в другом электронном оборудовании: цифровых фотоаппаратах, мобильных телефонах и т. п. Файловая система определяет формат содержимого и способ физического хранения информации, которую принято группировать в виде файлов. Конкретная файловая система определяет размер имен файлов (и каталогов), максимальный возможный размер файла и раздела, набор атрибутов файла. Некоторые файловые системы предоставляют сервисные возможности, например, разграничение доступа или шифрование файлов.

Файловая система связывает носитель информации с одной стороны и API для доступа к файлам — с другой.

Когда прикладная программа обращается к файлу, она не имеет никакого представления о том, каким образом расположена информация в конкретном файле, так же как и о том, на каком физическом типе носителя (CD, жёстком диске, магнитной ленте, блоке флеш-памяти или другом) он записан. Всё, что знает программа — это имя файла, его размер и атрибуты. Эти данные она получает от драйвера файловой системы. Именно файловая система

устанавливает, где и как будет записан файл на физическом носителе (например, жёстком диске).

С точки зрения операционной системы (ОС), весь диск представляет собой набор кластеров (как правило, размером 512 байт и больше)[1]. Драйверы файловой системы организуют кластеры в файлы и каталоги (реально являющиеся файлами, содержащими список файлов в этом каталоге). Эти же драйверы отслеживают, какие из кластеров в настоящее время используются, какие свободны, какие помечены как неисправные.

Однако файловая система не обязательно напрямую связана с физическим носителем информации. Существуют виртуальные файловые системы, а также сетевые файловые системы, которые являются лишь способом доступа к файлам, находящимся на удалённом компьютере (например samba).

Практически всегда файлы на дисках объединяются в каталоги.

Основными функциями файловой системы являются:

- размещение и упорядочивание на носителе данных в виде файлов;
- определение максимально поддерживаемого объема данных на носителе информации;
- создание, чтение и удаление файлов;
- назначение и изменение атрибутов файлов (размер, время создания и изменения, владелец и создатель файла, доступен только для чтения, скрытый файл, временный файл, архивный, исполняемый, максимальная длина имени файла и т.п.);
- определение структуры файла;
- поиск файлов;
- организация каталогов для логической организации файлов;
- защита файлов при системном сбое;
- защита файлов от несанкционированного доступа и изменения их содержимого.

В многопользовательских системах появляется ещё одна задача: защита файлов одного пользователя от несанкционированного доступа другого пользователя, а также обеспечение совместной работы с файлами, к примеру, при открытии файла одним из пользователей, для других этот же файл временно будет доступен в режиме «только чтение».

ФС позволяет оперировать не нулями и единицами, а более удобными и понятными объектами — файлами. Ради удобства в работе с файлами используются их символьные идентификаторы — имена. Само содержимое

файлов записано в кластеры ( clusters ) — мельчайшие единицы данных, которыми оперирует файловая система, размер их кратен 512 байтам (512 байт — размер сектора жесткого диска, минимальной единицы данных, которая считывается с диска или записывается на диск). Для организации информации кроме имени файла используются также каталоги (или папки), как некая абстракция, позволяющая группировать файлы по определенному критерию. По своей сути каталог — это файл, содержащий информацию о как бы вложенных в него каталогах и файлах.

Вся информация о файлах хранится в особых областях раздела (тома) — файловых справочниках. Структура этих справочников зависит от типа файловой системы. Справочник файлов позволяет ассоциировать числовые идентификаторы файлов и дополнительную информацию о них (дата изменения, права доступа, имя и т.д.) с непосредственным содержимым файла, хранящимся в другой области раздела (тома).

На жестких дисках компьютеров под управлением систем семейства Windows используются два типа файловых систем: FAT (FAT16 и FAT32) и NTFS.

## 9

### 9.1 Языки программирования

Язык программирования — формальный язык, предназначенный для записи компьютерных программ[1][2]. Язык программирования определяет набор лексических, синтаксических и семантических правил, определяющих внешний вид программы и действия, которые выполнит исполнитель (обычно — ЭВМ) под её управлением.

Языков программирования достаточно много. В основном они отличаются уровнем абстракции, которым можно описать алгоритм.

Как правило, язык программирования определяется не только через спецификации стандарта языка, формально определяющие его синтаксис и семантику, но и через воплощения (реализации) стандарта — программные средства, обеспечивающих трансляцию или интерпретацию программ на этом языке; такие программные средства различаются по производителю, марке и варианту (версии), времени выпуска, полноте воплощения стандарта, дополнительным возможностям; могут иметь определённые ошибки или

особенности воплощения, влияющие на практику использования языка или даже на его стандарт.

**Императивная** программа похожа на приказы (англ. *imperative* — приказ, повелительное наклонение), выражаемые повелительным наклонением в естественных языках, то есть представляют собой последовательность команд, которые должен выполнить компьютер.

**Декларативное** программирование — парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. Противоположностью декларативного является императивное программирование, описывающее на том или ином уровне детализации, как решить задачу и представить результат. В общем и целом, декларативное программирование идёт от человека к машине, тогда как императивное — от машины к человеку[уточнить].

## 9.2 Концепции процедурно-ориентированного программирования

Процедурное программирование является отражением архитектуры традиционных ЭВМ, которая была предложена Фон Нейманом в 1940-х годах. Теоретической моделью процедурного программирования служит абстрактная вычислительная система под названием машина Тьюринга (см опр в конце раздела).

Выполнение программы сводится к последовательному выполнению операторов с целью преобразования исходного состояния памяти, то есть значений исходных данных, в заключительное, то есть в результаты. Таким образом, с точки зрения программиста имеются программа и память, причем первая последовательно обновляет содержимое последней.

Процедурный язык программирования предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

**Машина Тьюринга** (абстрактный исполнитель) является расширением конечного автомата и, согласно тезису Чёрча — Тьюринга, способна имитировать всех исполнителей (с помощью задания правил перехода), каким-

либо образом реализующих процесс пошагового вычисления, в котором каждый шаг вычисления достаточно элементарен.

**Конечный автомат** - абстрактный автомат, число возможных внутренних состояний которого конечно.

**Абстрактный автомат** (в теории алгоритмов) — математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. На вход этому устройству поступают символы одного алфавита, на выходе оно выдаёт символы (в общем случае) другого алфавита.

### 9.3 Объектно-ориентированного программирование

Процедурно-ориентированная методология со временем перестала перекрывать весь объём задач, которые ставили перед программистами. Вычислительные и расчетно-алгоритмические задачи стали второстепенными, на первый план выступили задачи обработки и манипулирования данными. Стало очевидным, что традиционные методы процедурного программирования не способны справиться ни с растущей сложностью программ, ни с необходимостью их быстрой модернизации. В конце 80-х в начале 90-х годов была разработана методология объектно-ориентированного программирования (ООП), которая позволила по-новому взглянуть на процесс разработки программ и найти решение для целого комплекса проблем.

Базовыми понятиями ООП являются понятия класса и объекта. Класс - некоторая совокупность объектов (может быть абстрактная), которые имеют общий набор свойств и обладают одинаковым поведением. Объект - это экземпляр (представитель, элемент) соответствующего класса. Основными принципами ООП являются:

- **инкапсуляция** размещение в одном компоненте данных и методов, которые с ними работают. Также может означать скрытие внутренней реализации от других компонентов. Например, доступ к скрытой переменной может предоставляться не напрямую, а с помощью методов для чтения (геттер) и изменения (сеттер) её значения.; инкапсуляция связывает данные и методы в классе
- **наследование** свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью

зимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским. Новый класс — потомком, наследником, дочерним или производным классом.

- **полиморфизм** — это способность объекта использовать методы производного класса, который не существует на момент создания базового (*virtual* методы).
- **абстракция** Основная идея в том, чтобы представить объект минимальным набором полей и методов и при этом с достаточной точностью для решаемой задачи. Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой.;
- **посылка сообщений** — объект вызывает метод другого объекта.
- **повторное использование кода**

## 9.4 Функциональное программирование.

Это альтернатива процедурному программированию. Разница в том, что в процедурном нужно описать шаги КАК получить результат, а в функциональном – нужно описать результат.

```
function sumArray(values) {  
    return values.map(Number).reduce(  
        (previousValue, currentValue) => previousValue + currentValue,  
        0  
    );  
}  
  
sumArray([2, "5", 8]); // 15
```

Функциональный код отличается одним свойством: отсутствием побочных эффектов. Он не полагается на данные вне текущей функции, и не меняет данные, находящиеся вне функции. Все остальные «свойства» можно вывести из этого.

Функциональное программирование имеет преимущество в распараллеливании, тк функции чистые (детерминированы [при одних и тех же данных, один и тот же ответ] и нет побочных эффектов [нет зависимости от глобальных переменных]). Сложные задачи очень легко делятся на множество простых.

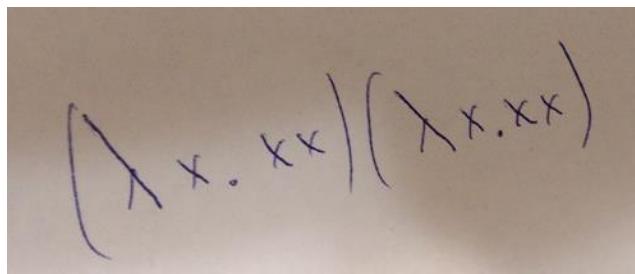
Функции высшего порядка (map) позволяют принимать и возвращать вектор.

Функция (reduce) – возвращает скаляр.

В функциональном программировании все (в случае чистых языков) или почти все (в случае «нечистых» — англ. impure — языков) данные в программе, как локальные, так и глобальные, являются неизменяемыми. С одной стороны, это существенно повышает стабильность программ за счёт упрощения формальной верификации программ. С другой, это затрудняет решение ряда задач (из которых наиболее часто отмечается задача реализации интерфейса пользователя, который в своей сути представляет собой изменяемое состояние), что вынуждает усложнять системы типов языка — например, монадами или уникальными типами.

При копировании данных при передачи из функции в функцию может использоваться много места при работе с биг датой. Поэтому иногда используется copy on write.

Может быть забавная штука — Куайн (компьютерная программа, которая выдаёт на выходе точную копию своего исходного текста).



## 9.5 Логическое программирование

Центральным понятием в логическом программировании является отношение. Программа представляет собой совокупность определенных отношений между объектами (в терминах, условий или ограничений) и цели (запроса). Процесс выполнения программы трактуется как процесс установления общезначимости логической формулы, построенной из программы по правилам, установленным семантикой того или иного языка.

Логическое программирование — парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

Самым известным языком логического программирования является Prolog.

Логика высказываний, пропозициональная логика (лат. *propositio* — «высказывание»[1]) или исчисление высказываний[2] — это раздел символической логики, изучающий сложные высказывания, образованные из простых, и их взаимоотношения. В отличие от логики предикатов, пропозициональная логика не рассматривает внутреннюю структуру простых высказываний, она лишь учитывает, с помощью каких союзов и в каком порядке простые высказывания сочленяются в сложные[3].

### Логические знаки высказываний

Символ	Значение
$\neg$	Знак <b>отрицания</b>
$\wedge$ или &	Знак <b>конъюнкции</b> («И»)
$\vee$	Знак <b>дизъюнкции</b> («включающее ИЛИ»)
$\oplus$ или $\dot{\vee}$	Знак <b>строгой дизъюнкции</b> («исключающее ИЛИ»)
$\rightarrow$	Знак <b>импликации</b>
$\leftrightarrow$ или $\sim$ или $\equiv$	Знак <b>эквивалентности</b>



$$\neg A \vee B.$$

Одним из возможных вариантов (гильбертовской) аксиоматизации логики высказываний является следующая система аксиом:

- $$A_1 : A \rightarrow (B \rightarrow A);$$
- $$A_2 : (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C));$$
- $$A_3 : A \wedge B \rightarrow A;$$
- $$A_4 : A \wedge B \rightarrow B;$$
- $$A_5 : A \rightarrow (B \rightarrow (A \wedge B));$$
- $$A_6 : A \rightarrow (A \vee B);$$
- $$A_7 : B \rightarrow (A \vee B);$$
- $$A_8 : (A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow ((A \vee B) \rightarrow C));$$
- $$A_9 : \neg A \rightarrow (A \rightarrow B);$$
- $$A_{10} : (A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A);$$
- $$A_{11} : A \vee \neg A.$$

вместе с единственным правилом:

$$\frac{A \quad (A \rightarrow B)}{B} \text{ (*Modus ponens*)}$$

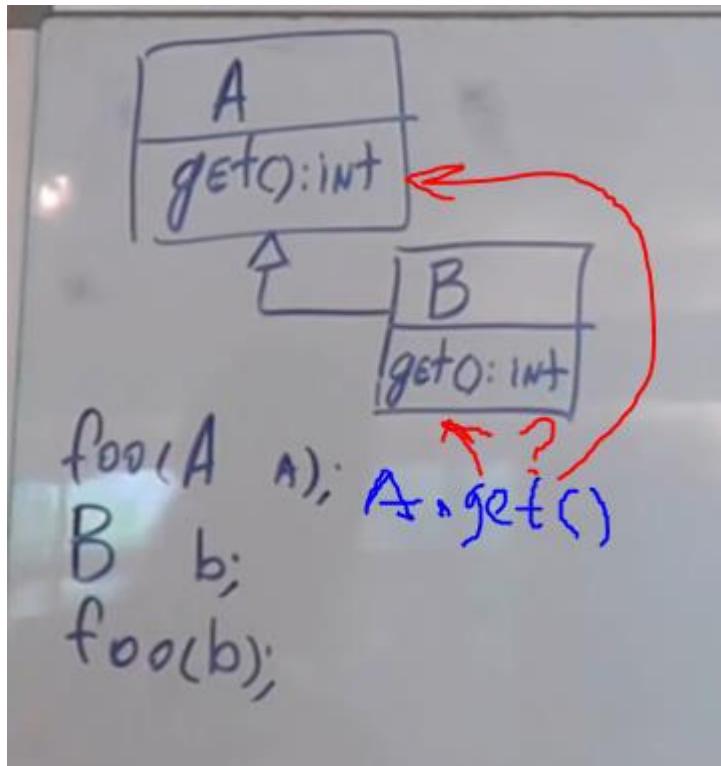
## Quicksort [ edit ]

The [quicksort](#) sorting algorithm, relating a list to its sorted version:

```
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    (   X < Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).

quicksort([])      --> [].
quicksort([X|Xs]) -->
    { partition(Xs, X, Smaller, Bigger) },
    quicksort(Smaller), [X], quicksort(Bigger).
```

## 9.6 Раннее (статическое) и позднее (динамическое) связывание



Раннее – компилятор во время создания программы гарантирует, какая функция будет вызвана.

Позднее – он не знает, какую ф-ю вызвать. Он ищет функцию `get()` во время вызова функции. В си – `virtual` позволяет получить позднее связывание.

## 9.7 Статическая и динамическая типизация.

По сути это одно и тоже, но если статическая/динамическая отвечает на вопрос "когда делается", то явная/неявная — "кто делает"

### 9.7.1 Статически / динамики типизированные языки

**Статическая типизация** определяется тем, что конечные типы переменных и функций устанавливаются на этапе компиляции. Т.е. уже компилятор на 100% уверен, какой тип где находится. В динамической типизации все типы выясняются уже во время выполнения программы.

Статически типизированные языки ограничивают типы переменных: язык программирования может знать, например, что `x` — это `Integer`. В этом случае программисту запрещается делать `x = true`, это будет некорректный код.

**Динамически** типизированные языки не требуют указывать тип, но и не определяют его сами. Типы переменных неизвестны до того момента, когда у них есть конкретные значения при запуске.

Динамически типизированные языки помечают значения типами: язык знает, что 1 это integer, 2 это integer, но он не может знать, что переменная x всегда содержит integer.

Статическая: C, Java, C#;

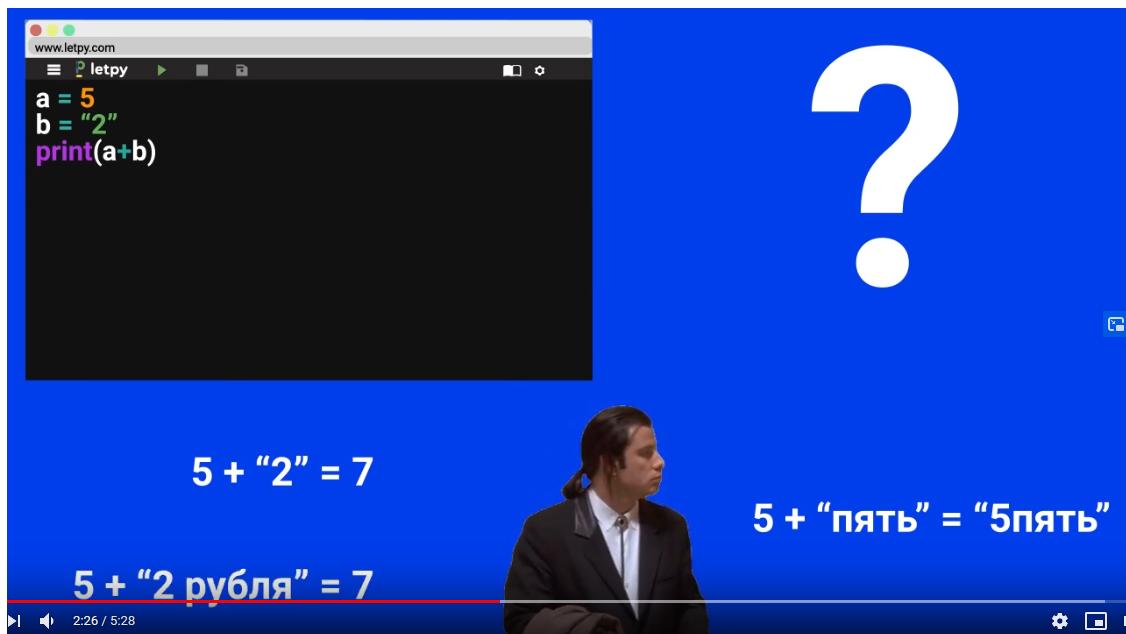
Динамическая: Python, JavaScript, Ruby.

### 9.7.2 Сильная / слабая типизация



Сильная типизация выделяется тем, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например нельзя вычесть из строки множество. Языки со слабой типизацией выполняют множество неявных преобразований автоматически, даже если может произойти потеря точности или преобразование неоднозначно.

**Сильная** типизация – когда компилятор явно не может понять, что делать с данными разного типа, тк явно они не преобразуются.



**Слабая** типизация – когда компилятор может сам попробовать преобразовать 2 переменные разного типа в один общий тип. Например `c++ 0 == false == nullptr`.

Сильная: Java, Python, Haskell, Lisp;

Слабая: C, JavaScript, Visual Basic, PHP.

### 9.7.3 Явная / неявная типизация

Явно-тиปизированные языки отличаются тем, что тип новых переменных / функций / их аргументов нужно задавать явно. Соответственно языки с неявной типизацией перекладывают эту задачу на компилятор / интерпретатор.

В новом стандарте языка C++, названном C++11 (ранее назывался C++0x), было введено ключевое слово `auto`, благодаря которому можно заставить компилятор вывести тип, исходя из контекста.

Явная: C++, D, C#

Неявная: PHP, Lua, JavaScript, python

### 9.7.4 Итог

Также нужно заметить, что все эти категории пересекаются, например язык С имеет статическую слабую явную типизацию, а язык Python — динамическую сильную неявную.

JavaScript	- Динамическая	Слабая	Неявная
Ruby	- Динамическая	Сильная	Неявная
Python	- Динамическая	Сильная	Неявная
Java	- Статическая	Сильная	Явная
PHP	- Динамическая	Слабая	Неявная
C	- Статическая	Слабая	Явная
C++	- Статическая	Слабая	Явная
Perl	- Динамическая	Слабая	Неявная
Objective-C	- Статическая	Слабая	Явная
C#	- Статическая	Сильная	Явная
Haskell	- Статическая	Сильная	Неявная
Common Lisp	- Динамическая	Сильная	Неявная
D	- Статическая	Сильная	Явная
Delphi	- Статическая	Сильная	Явная

# 10

Для общего ознакомления смотри books/lab6.pdf

## 10.1 Понятие о методах трансляции

Трансля́тор — программа или техническое средство, выполняющее трансляцию программы.

Трансля́ция прого́раммы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т. д.

## 10.2 Лексический анализ

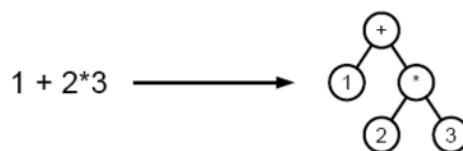
В информатике лексический анализ («токенизация», от англ. tokenizing) — процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы (последовательность допустимых символов языка программирования, имеющая смысл для транслятора) — с целью получения на выходе идентифицированных последовательностей, называемых «токенами».

Как правило, лексический анализ производится с точки зрения определённого формального языка или набора языков. Язык, а точнее, его

грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

### 10.3 Синтаксический анализ

Синтаксический анализ (или разбор, жарг. пárсинг ← англ. parsing) в лингвистике и информатике — процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево).



В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки.

Простейший способ реагирования на некорректную входную цепочку лексем — завершить синтаксический анализ и вывести сообщение об ошибке. Однако часто оказывается полезным найти за одну попытку синтаксического анализа как можно больше ошибок. Именно так ведут себя трансляторы большинства распространённых языков программирования.

Таким образом перед обработчиком ошибок синтаксического анализатора стоят следующие задачи:

- он должен ясно и точно сообщать о наличии ошибок;
- он должен обеспечивать быстрое восстановление после ошибки, чтобы продолжать поиск других ошибок;
- он не должен существенно замедлять обработку корректной входной цепочки.

#### 10.3.1 Восстановление в режиме паники

При обнаружении ошибки синтаксический анализатор пропускает входные лексемы по одной, пока не будет найдена одна из специально определённого множества синхронизирующих лексем. Обычно такими лексемами являются разделители, например:

- ;

- )
- }

Набор синхронизирующих лексем должен определять разработчик анализируемого языка. При такой стратегии восстановления может оказаться, что значительное количество символов будут пропущены без проверки на наличие дополнительных ошибок. Данная стратегия восстановления наиболее проста в реализации.

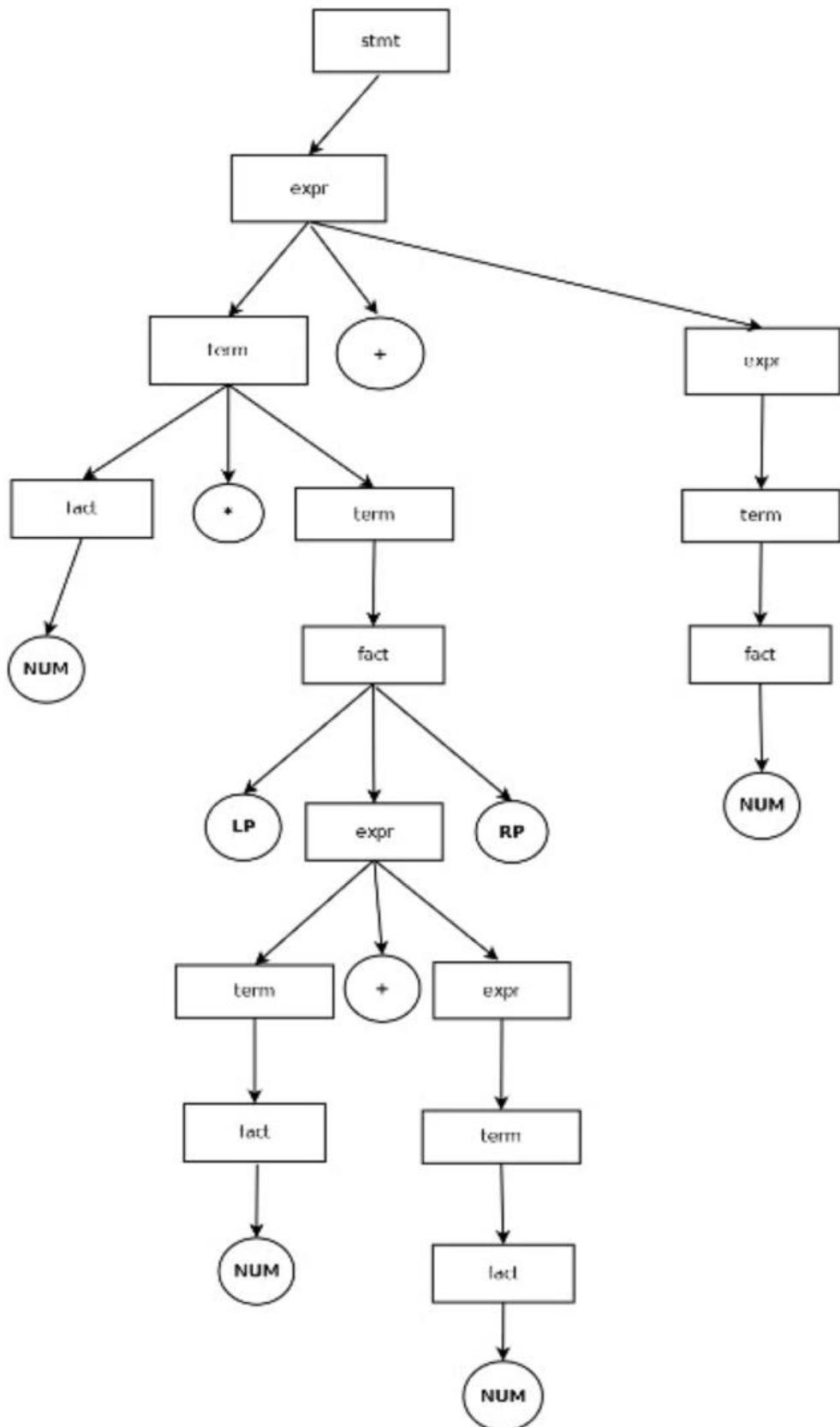
### *10.3.2 Восстановление на уровне фразы*

Иногда при обнаружении ошибки синтаксический анализатор может выполнить локальную коррекцию входного потока так, чтобы это позволило ему продолжать работу. Например, перед точкой с запятой, отделяющей различные операторы в языке программирования, синтаксический анализатор может закрыть все ещё не закрытые круглые скобки. Это более сложный в проектировании и реализации способ, однако в некоторых ситуациях, он может работать значительно лучше восстановления в режиме паники. Естественно, данная стратегия бессильна, если настоящая ошибка произошла до точки обнаружения ошибки синтаксическим анализатором.

### *10.3.3 Нисходящий синтаксический анализ*

$$\begin{array}{l} \text{stint} \rightarrow \text{expr} \mid \epsilon \\ \text{expr} \rightarrow \text{term} \mid \text{term} + \text{expr} \\ \text{term} \rightarrow \text{fact} \mid \text{fact} * \text{term} \\ \text{fact} \rightarrow \text{NUM} \mid ( \text{expr} ) \end{array}$$

$$3^*(1+2)+4$$



## 10.4 Семантический анализ

Смысл (семантику) каждой такой синтаксической конструкции входного языка можно определить, исходя из ее типа, а тип определяется синтаксическим

анализатором на основе грамматики входного языка. Примерами типов синтаксических конструкций могут служить операторы цикла, условные операторы, операторы выбора и т. д. Одни и те же типы синтаксических конструкций характерны для различных языков программирования, при этом они различаются синтаксисом (который задается грамматикой языка), но имеют схожий смысл (который определяется семантикой). В зависимости от типа синтаксической конструкции выполняется генерация кода результирующей программы, соответствующего данной синтаксической конструкции. Для семантически схожих конструкций различных входных языков программирования может порождаться типовой результирующий код.

## 10.5 Основные алгоритмы генерации объектного кода

Генерация объектного кода – это перевод компилятором внутреннего представления исходной программы в результирующую объектную программу на языке ассемблера или непосредственно на машинном языке (машинных кодах). Генерация объектного кода выполняется после того, как выполнен синтаксический анализ программы и все необходимые действия по подготовке к генерации кода: распределено адресное пространство под функции и переменные, проверено соответствие имен и типов переменных, констант и функций в синтаксических конструкциях исходной программы и т.д.

Теперь рассмотрим общий вариант алгоритма генерации кода, который получает на входе дерево вывода (построенное в результате синтаксического разбора) и создает по нему фрагмент объектного кода результирующей программы. Алгоритм должен выполнить следующую последовательность действий:

- построить последовательность триад на основе дерева вывода 10.5.1;
- выполнить оптимизацию кода методом свертки для линейных участков результирующей программы 10.5.2;
- выполнить оптимизацию кода методом исключения лишних операций для линейных участков результирующей программы 10.5.3;
- преобразовать последовательность триад в последовательность команд на языке ассемблера (полученная последовательность команд и будет результатом выполнения алгоритма).

Алгоритм преобразования триад в команды языка ассемблера – это единственная машинно-зависимая часть общего алгоритма. При преобразовании

компилятора для работы с другим результирующим объектным кодом потребуется изменить только эту часть, при этом все алгоритмы оптимизации и внутреннее представление программы останутся неизменными.

### *10.5.1 Многоадресный код с неявно именуемым результатом (триады)*

Триады представляют собой запись операций в форме из трех составляющих: операция и два операнда. Например, в строковой записи триады могут иметь вид: <операция>(<операнд1>,<операнд2>). Особенностью триад является то, что один или оба операнда могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады. Поэтому триады при записи последовательно нумеруют для удобства указания ссылок одних триад на другие (в реализации компилятора в качестве ссылок можно использовать не номера триад, а непосредственно ссылки в виде указателей – тогда при изменении нумерации и порядка следования триад менять ссылки не требуется).

Например, выражение  $A = B \cdot C + D - B \cdot 10$ , записанное в виде триад, будет иметь вид:

```
1: * ( B. C )
2: + ( ^1. D )
3: * ( B. 10 )
4: - ( ^2. ^3 )
5: := ( A. ^4 )
```

Здесь операции обозначены соответствующими знаками (при этом присваиваниетакже является операцией), а знак  $\wedge$ означает ссылку операнда одной триады на результат другой. Триады представляют собой линейную последовательность команд. При вычислении выражения, записанного в форме триад, они вычисляются одна за другой последовательно. Каждая триада в последовательности вычисляется так: операция, заданная триадой, выполняется над операндами, а если в качестве одного из operandов (или обоих operandов) выступает ссылка на другую триаду, то берется результат вычисления той триады. Результат вычисления триады нужно сохранять во временной памяти, так как он может быть затребован последующими триадами.

### *10.5.2 Свертка объектного кода*

Свертка объектного кода – это выполнение во время компиляции тех операций исходной программы, для которых значения операндов уже известны. Нет необходимости многократно выполнять эти операции в результате программы – вполне достаточно один раз выполнить их при компиляции.

Простейший вариант свертки – выполнение в компиляторе операций, операндами которых являются константы. Несколько более сложен процесс определения тех операций, значения которых могут быть известны в результате выполнения других операций. Для этой цели при оптимизации линейных участков программы используется специальный алгоритм свертки объектного кода.

Алгоритм свертки триад последовательно просматривает триады линейного участка и для каждой триады делает следующее:

1. Если operand есть переменная, которая содержится в таблице Т, то operand заменяется на соответствующее значение константы.
2. Если operand есть ссылка на особую триаду типа С(К,0), то operand заменяется на значение константы К.
3. Если все operandы триады являются константами, то триада может быть свернута. Тогда данная триада выполняется и вместо нее помещается особая триада вида С(К,0), где К – константа, являющаяся результатом выполнения свернутой триады. (При генерации кода для особой триады объектный код не порождается, а потому она в дальнейшем может быть просто исключена.)
4. Если триада является присваиванием типа А:=В, тогда:
  - если В – константа, то А со значением константы заносится в таблицу Т (если там уже было старое значение для А, то это старое значение исключается);
  - если В – не константа, то А вообще исключается из таблицы Т, если оно там есть.

Рассмотрим пример выполнения алгоритма. Усть фрагмент исходной программы (записанной на языке типа Pascal) имеет вид

```
I := 1 + 1;  
I := 3;  
J := 6*I + I;
```

Ее внутреннее представление в форме триад будет иметь вид

```

1: + (1, 1)
2: := (I, ^1)
3: := (I, 3)
4: * (6, I)
5: + (^4, I)
6: := (J, ^5)

```

Процесс выполнения алгоритма свертки показан в табл

Триада	Шаг 1	Шаг 2	Шаг 3	Шаг 4	Шаг 5	Шаг 6
1	C (2, 0)					
2	:= (I, ^1)	:= (I, 2)				
3	:= (I, 3)					
4	* (6, I)	* (6, I)	* (6, I)	C (18, 0)	C (18, 0)	C (18, 0)
5	+ (^4, I)	+ (^4, I)	+ (^4, I)	+ (^4, I)	C (21, 0)	C (21, 0)
6	:= (J, ^5)	:= (J, 21)				
T	( , )	(I, 2)	(I, 3)	(I, 3)	(I, 3)	(I, 3) (J, 21)

Если исключить особые триады типа C(K,0) (которые не порождают объектного кода), то в результате выполнения свертки получим следующую последовательность триад:

```

1: := (I, 2)
2: := (I, 3)
3: := (J, 21)

```

Алгоритм свертки объектного кода позволяет исключить из линейного участка программы операции, для которых на этапе компиляции уже известен результат. За счет этого сокращается время выполнения, а также объем кода результирующей программы.

### 10.5.3 Исключение лишних операций

#### 10.5.3.1 Триады

Исключение избыточных вычислений (лишних операций) заключается в нахождении и удалении из объектного кода операций, которые повторно обрабатывают одни и те же операнды.

Алгоритм исключения лишних операций просматривает операции в порядке их следования. Так же как и алгоритму свертки, алгоритму исключения

лишних операций проще всего работать с триадами, потому что они полностью отражают взаимосвязь операций.

Рассмотрим алгоритм исключения лишних операций для триад.

Handwritten notes:

$$dep(\text{var}) = \begin{cases} i & \text{если дано присвоение} \\ 0 & \text{иначе} \end{cases}$$
$$dep(\text{operator}(\text{arg1}, \text{arg2})) = 1 + \max(dep(\text{arg1}), dep(\text{arg2}))$$

Чтобы следить за внутренней зависимостью переменных и триад, алгоритм присваивает им некоторые значения, называемые числами зависимости, по следующим правилам:

- изначально для каждой переменной ее число зависимости равно 0, так как в начале работы программы значение переменной не зависит ни от какой триады;
- после обработки  $i$ -й триады, в которой переменной A присваивается некоторое значение, число зависимости A ( $dep(A)$ ) получает значение  $i$ , так как значение A теперь зависит от данной  $i$ -й триады;
- при обработке  $i$ -й триады ее число зависимости ( $dep(i)$ ) принимается равным  $1 + \max(\text{число зависимости arg1}, \text{число зависимости arg2})$ .

Таким образом, при использовании чисел зависимости триад и переменных можно утверждать, что если  $i$ -я триада идентична  $j$ -й триаде ( $j < i$ ), то  $i$ -я триада считается лишней в том и только в том случае, когда  $dep(i) = dep(j)$ .

Алгоритм исключения лишних операций использует в своей работе триады особого вида  $SAME(j, 0)$ . Если такая триада встречается в позиции с номером  $i$ , то это означает, что в исходной последовательности триад некоторая триада I идентична триаде j.

Алгоритм исключения лишних операций последовательно просматривает триады линейного участка. Он состоит из следующих шагов, выполняемых для каждой триады:

1. Если какой-то operand триады ссылается на особую триаду вида  $SAME(j, 0)$ , то он заменяется на ссылку на триаду с номером  $j (^j)$ .
2. Вычисляется число зависимости текущей триады с номером  $i$ , исходя из чисел зависимости ее operandов.

3. Если в просмотренной части списка триад существует идентичная j-я триада, причем  $j < i$  и  $\text{dep}(i) = \text{dep}(j)$ , то текущая триада заменяется на триаду особого вида  $\text{SAME}(j, 0)$ .
4. Если текущая триада есть присваивание, то вычисляется число зависимости соответствующей переменной.

Рассмотрим работу алгоритма на примере:

$D := D + C * B;$

$A := D + C * B;$

$C := D + C * B;$

Этому фрагменту программы будет соответствовать следующая последовательность триад

```

1: * (C, B)
2: + (D, ^1)
3: := (D, ^2)
4: * (C, B)
5: + (D, ^4)
6: := (A, ^5)
7: * (C, B)
8: + (D, ^7)
9: := (C, ^8)

```

Видно, что в данном примере некоторые операции вычисляются дважды над одними и теми же operandами, а значит, они являются лишними и могут быть исключены. Работа алгоритма исключения лишних операций отражена в табл

+ triad	dep (var)				dep(tired)
	A	B	C	D	
	0	0	0	0	
1) * C, B	0	0	0	0	1
2) + D ^ 1	0	0	0	0	2
3) := D ^ 2	0	0	0	3	3
4) * C, B	0	0	0	3	1
5) + D ^ 4	0	0	6	3	4
6) := A ^ 5	6	0	0	3	5

Обрабатываемая триада i	Числа зависимости переменных				Числа зависимости триад dep(i)	Триады, полученные после выполнения алгоритма
	A	B	C	D		
1) * (C, B)	0	0	0	0	1	1) * (C, B)
2) + (D, ^1)	0	0	0	0	2	2) + (D, ^1)
3) := (D, ^2)	0	0	0	3	3	3) := (D, ^2)
4) * (C, B)	0	0	0	3	1	4) SAME (1, 0)
5) + (D, ^4)	0	0	0	3	4	5) + (D, ^1)
6) := (A, ^5)	6	0	0	3	5	6) := (A, ^5)
7) * (C, B)	6	0	0	3	1	7) SAME (1, 0)
8) + (D, ^7)	6	0	0	3	4	8) SAME (5, 0)
9) := (C, ^8)	6	0	9	3	5	9) := (C, ^5)

Теперь, если исключить триады особого вида SAME(j,0), то в результате выполнения алгоритма получим следующую последовательность триад:

```
1: * (C, B)
2: + (D, ^1)
3: := (D, ^2)

4: + (D, ^1)
5: := (A, ^4)
6: := (C, ^4)
```

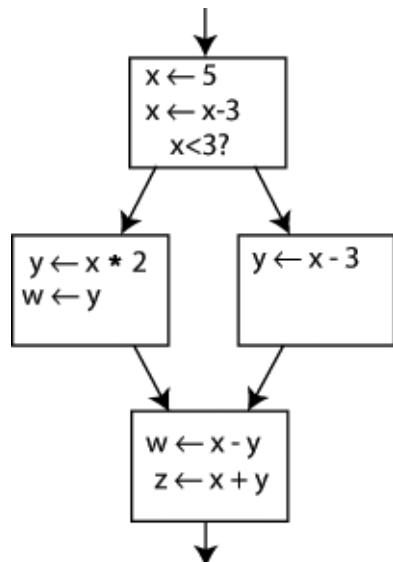
Обратите внимание, что в итоговой последовательности изменилась нумерация триад и номера в ссылках одних триад на другие. Если в компиляторе в качестве ссылок использовать не номера триад, а непосредственно указатели на них, то изменения ссылок в таком варианте не потребуется.

Алгоритм исключения лишних операций позволяет избежать повторного выполнения одних и тех же операций над одними и теми же operandами. В результате оптимизации по этому алгоритму сокращается и время выполнения, и объем кода результирующей программы.

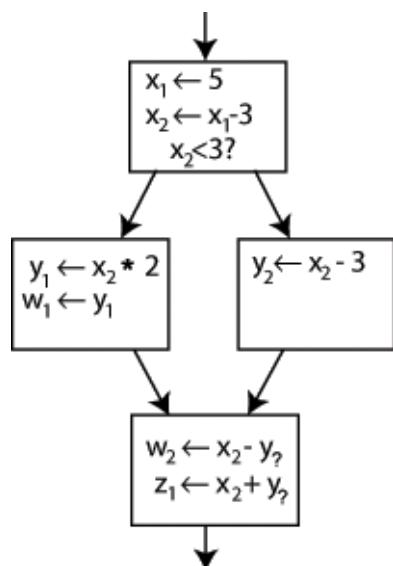
#### 10.5.3.2 SSA

SSA (англ. Static single assignment form) — промежуточное представление, используемое компиляторами, в котором каждой переменной значение присваивается лишь единожды. Переменные исходной программы разбиваются на версии, обычно с помощью добавления суффикса, таким образом, что каждое присваивание осуществляется уникальной версии переменной. В SSA-представлении DU-цепи (англ. def-use) заданы явно и содержат единственный элемент.

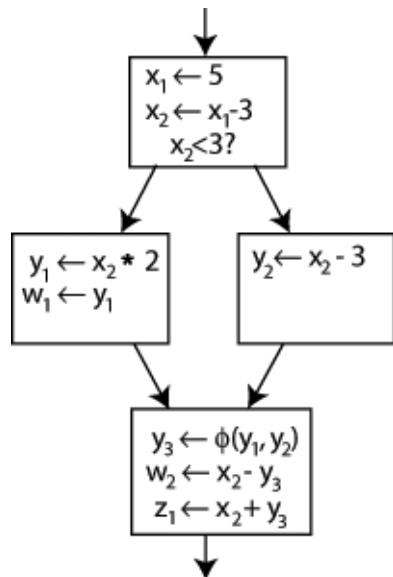
Перевод обычного программного кода в SSA-представление достигается путём замены в каждой операции присваивания переменной из левой части на новую переменную. Для каждого использования значения переменной исходное имя заменяется на имя «версии», соответствующей нужному базовому блоку. Рассмотрим следующий график потока управления:



В соответствии с определением SSA создадим вместо переменной  $x$  две новые переменные  $x_1$  и  $x_2$ . Каждой из них значение будет присвоено ровно один раз. Аналогичным образом заменим остальные переменные, после чего получим следующий граф:



Пока остаётся неясным, какое значение  $y$  будет использоваться в нижнем блоке. Там имя  $y$  может означать как  $y_1$ , так и  $y_2$ . Для того, чтобы разрешить неоднозначности такого рода, в SSA введена специальная Ф-функция. Эта функция создаёт новую версию переменной  $y$ , которой будет присвоено значение либо из  $y_1$ , либо из  $y_2$ , в зависимости от того, из какой ветви перешло управление.



При этом использовать  $\Phi$ -функцию для переменной  $x$  не нужно, так как лишь одна версия  $x$  (а именно,  $x_2$ ) «достигает» последнего блока.

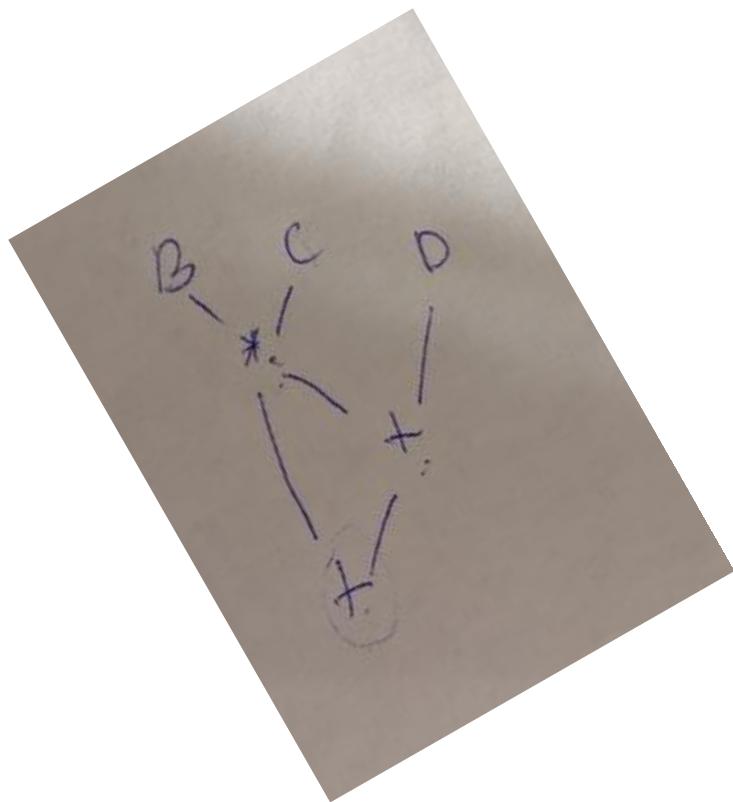
$\Phi$ -функция в действительности не реализована; она представляет собой лишь указание компилятору хранить все переменные, перечисленные в списке её аргументов, в одном и том же месте в памяти (или регистре).

Более общий вопрос состоит в том, можно ли по заданному графу потока управления понять, где и для каких переменных в SSA-представление нужно вставить  $\Phi$ -функции? Ответ на этот вопрос можно получить с помощью доминаторов.

Пример:

```

D := D + C*B;
A := D + C*B;
C := D + C*B;
  
```



#### *10.5.4 Синтаксически управляемый (СУ) перевод*

Идея СУ-перевода основана на том, что синтаксис и семантика языка взаимосвязаны. Это значит, что смысл предложения языка зависит от синтаксической структуры этого предложения.

Например, смысл предложения русского языка зависит от входящих в него частей речи (подлежащего, сказуемого, дополнений и др.) и от взаимосвязи между ними. Однако естественные языки допускают неоднозначности в грамматиках—отсюда происходят различные двусмысленные фразы, значение которых человек обычно понимает из того контекста, в котором эти фразы встречаются (и то он не всегда может это сделать). В языках программирования неоднозначности в грамматиках исключены, поэтому любое предложение языка имеет четко определенную структуру и однозначный смысл, напрямую связанный с этой структурой.

Грубо говоря, идея СУ-перевода заключается в том, что каждому правилу входного языка компилятора сопоставляется одно или несколько (или ни одного) правил выходного языка в соответствии с семантикой входных и выходных правил. То есть при сопоставлении надо выбирать правила выходного языка, которые несут тот же смысл, что и правила входного языка.

### *10.5.5 Шлак*

Генерацию кода можно считать функцией, определенной на синтаксическом дереве, построенном в результате синтаксического анализа. В идеале компилятор должен выполнить синтаксический анализ всей входной программы, затем провести ее семантический анализ, после чего приступить к подготовке генерации и непосредственно генерации кода. Однако такая схема работы компилятора практически никогда не применяется.

Дело в том, что в общем случае ни один семантический анализатор и ни один компилятор не способны проанализировать и оценить смысл всей исходной программы в целом. Формальные методы анализа семантики применимы только к очень незначительной части возможных исходных программ. Поэтому у компилятора нет практической возможности порождать эквивалентную результирующую программу на основе всей исходной программы.

Как правило, компилятор выполняет генерацию результирующего кода поэтапно, на основе законченных синтаксических конструкций входной программы. Компилятор выделяет законченную синтаксическую конструкцию из текста исходной программы, порождает для нее фрагмент результирующего кода и помещает его в текст результирующей программы. Затем он переходит к следующей синтаксической конструкции. Так продолжается до тех пор, пока не будет разобрана вся исходная программа.

В качестве анализируемых законченных синтаксических конструкций выступают блоки операторов, описания процедур и функций. Их конкретный состав зависит от входного языка и реализации компилятора.

## **10.6 Машинно-ориентированные языки (ассемблеры)**

Язык ассемблера (англ. assembly language) — машинно-ориентированный язык программирования низкого уровня. Его команды прямо соответствуют отдельным командам машины или их последовательностям, также он может предоставлять дополнительные возможности облегчения программирования, такие как макрокоманды, выражения, средства обеспечения модульности программ. Может рассматриваться как автокод (см. ниже), расширенный конструкциями языков программирования высокого уровня. Является существенно платформо-зависимым. Языки ассемблера для различных аппаратных платформ несовместимы, хотя могут быть в целом подобны.

В русском языке может именоваться просто «ассемблером» (типичны выражения типа «писать программу на ассемблере»), что, строго говоря, неверно, так как ассемблером именуется утилита трансляции программы с языка ассемблера в машинный код компьютера.

Автокод — язык программирования, предложения которого по своей структуре в основном подобны командам и обрабатываемым данным конкретного машинного языка[3].

Язык ассемблера — система обозначений, используемая для представления в удобно читаемой форме программ, записанных в машинном коде. Язык ассемблера позволяет программисту пользоваться алфавитными мнемоническими кодами операций, по своему усмотрению присваивать символические имена регистрам ЭВМ и памяти, а также задавать удобные для себя схемы адресации (например, относительную или абсолютную). Кроме того, он позволяет использовать различные системы счисления (например, десятичную или шестнадцатеричную) для представления числовых констант и даёт возможность помечать строки программы метками с символическими именами с тем, чтобы к ним можно было обращаться (по именам, а не по адресам) из других частей программы;.

Перевод программы на языке ассемблера в исполнимый машинный код (вычисление выражений, раскрытие макрокоманд, замена мнемоник собственно машинными кодами и символьных адресов на абсолютные или относительные адреса) производится ассемблером — программой-транслятором, которая и дала языку ассемблера его название.

## **10.7 Макросредства, макровызовы, языки макроопределений, условная макрогенерации принципы реализации**

### *10.7.1 Макросредства*

Макрокоманда, или макрос — программный алгоритм действий, записанный пользователем. Часто макросы применяют для выполнения рутинных действий. А также макрос — это символьное имя в шаблонах, заменяемое при обработке препроцессором на последовательность символов.

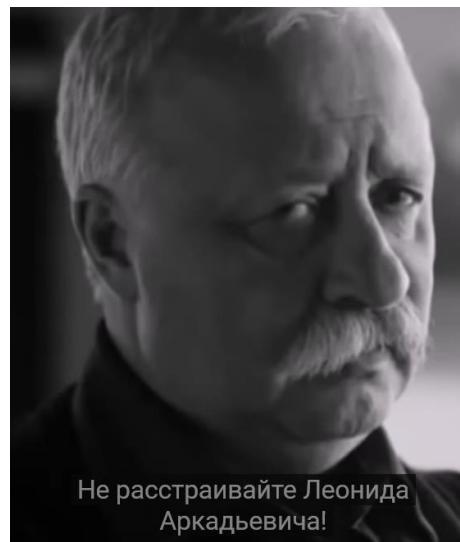
В языках ассемблера, а также в некоторых других языках программирования, макрос — символьное имя, заменяемое при обработке препроцессором на последовательность программных инструкций.

Макросы вещь достаточно спорная, тк их частое использование говорит о некоторых недостатков языка, но эта возможность позволяет пользователю делать собственные конструкции, которые заранее в языке небыли предусмотрены.

На примере фреймворка Qt, которым я пользуюсь, на макросах реализовано очень много, например кроссплатформенность. Но для восприятия эти алгоритмы часто выглядят как магия. В неумелых руках макросы могут очень сильно усложнить код.

Но на примере микроконтроллеров (Arduino 2016)

- номер пина можно задefайнить, тк эта переменная находится во флеше,
- нужен код, который переписывает из флеша в рам
- удлиняются инструкции, тк числа грусятся из рам, а не напрямую.
- Адрес переменной каждый раз достается и складывается в стек в цикле loop



### 10.7.2 Макровызовы

Макроопределение может иметь параметры. Например:

```
#define add(a, b) a+b
```

Но злоупотребление может привести к такому

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
#define DISP = 4
int xx = 0;

void f() {
    int xx = 0;
    xx = SQUARE(xx+2);
    INCR_xx;
    if (a-DISP==b) {
        // ...
    }
}
```

### *10.7.3 Условная макрогенерация*

Условная макрогенерация позволяет компилировать или пропускать часть программы в зависимости от выполнения некоторого условия. Это удобно, когда нужно писать кроссплатформенный код, и по-разному обращаться к системным функциям (чтение/запись файла) на разных ос.

## **10.8 Системы программирования, типовые компоненты**

Система программирования - это набор специализированных программных продуктов, которые являются инструментальными средствами разработчика.

Программные продукты данного класса поддерживают все этапы процесса программирования, отладки и тестирования создаваемых программ.

### *10.8.1 Редактор текста*

это программа для ввода и модификации текста.

### *10.8.2 Трансляторы*

Трансляторы предназначены для преобразования программ, написанных на языках программирования, в программы на машинном языке. Программа, подготовленная на каком-либо языке программирования, называется исходным модулем. В качестве входной информации трансляторы применяют исходные модули и формируют в результате своей работы объектные модули, являющиеся входной информацией для редактора связей. Объектный модуль содержит текст программы на машинном языке и дополнительную информацию, обеспечивающую настройку модуля по месту его загрузки и объединение этого

модуля с другими независимо оттранслированными модулями в единую программу.

Трансляторы делятся на два класса: компиляторы и интерпретаторы. Компиляторы переводят весь исходный модуль на машинный язык. Интерпретатор последовательно переводит на машинный язык и выполняет операторы исходного модуля

#### *10.8.3 Компоновщик, или редактор связей*

Компоновщик, или редактор связей - системная обрабатывающая программа, редактирующая и объединяющая объектные (ранее отраслированные) модули в единые загрузочные, готовые к выполнению программные модули. Загрузочный модуль может быть помещен ОС в основную память и выполнен.

#### *10.8.4 Отладчик*

Отладчик позволяет управлять процессом исполнения программы, является инструментом для поиска и исправления ошибок в программе. Базовый набор функций отладчика включает:

- пошаговое выполнение программы (режим трассировки) с отображением результатов,
- остановка в заранее определенных точках,
- возможность остановки в некотором месте программы при выполнении некоторого условия;
- изображение и изменение значений переменных.

#### *10.8.5 Загрузчик*

Загрузчик - системная обрабатывающая программа. Загрузчик помещает объектные и загрузочные модули в оперативную память, объединяет их в единую программу, корректирует перемещаемые адресные константы с учетом фактического адреса загрузки и передает управление в точку входа созданной программы.

## **11.1 Принципы модульного, компонентного, объектно-ориентированного проектирования**

### *11.1.1 Принципы модульного проектирования*

Модульное программирование — это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определённым правилам. Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

Модуль — функционально законченный фрагмент программы. Во многих языках (но далеко не обязательно) оформляется в виде отдельного файла с исходным кодом или поименованной непрерывной её части.

Принцип модульности является средством упрощения задачи проектирования ПС и распределения процесса разработки ПС между группами разработчиков. При разбиении ПС на модули для каждого модуля указывается реализуемая им функциональность, а также связи с другими модулями.[2] Удобство использования модульной архитектуры заключается в возможности обновления (замены) модуля, без необходимости изменения остальной системы.

### *11.1.2 Принципы компонентного проектирования*

Суть компонентно-ориентированного программирования (далее КОП) сводится к возможности контролировать взаимодействие проектируемых и выполняемых модулей на предмет согласованности информационных структур.

Несмотря на свою относительную молодость, КОП имеет свои особенности, которые регулируют не только особенности языка, но и всей экосистемы КОП. К таким отличительным чертам следует отнести:

- Чётко выраженную ориентированность на модули. Модуль является основной структурной единицей.
- Раздельная компиляция модулей. Это приводит к сбережению вычислительных и временных ресурсов.
- Строгая типизация, как внутри модуля, так и между модулями. Обеспечивает надёжную работу компонентов в целом.

- Неизбежность динамической сборки мусора. Для компилируемых языков это важный и необычный механизм.
- Строгое разделение частей модулей, предназначенных для взаимодействия с другими модулями, и скрытые части только для работы внутри модуля.

Основу объектно-компонентного метода составляет четырехуровневое моделирование объектных систем с помощью логико-математических операций. На первом уровне проводится декомпозиция предметной области на объекты. На следующих уровнях определяются их внешние характеристики и формируется объектный граф  $G$ , в вершинах которого находятся объекты, а дуги задают связи (интерфейсы) и данные, которыми обмениваются с другими объектами. На поведенческом уровне выявляется поведение объектов и строится модель поведения. Объекты графа переводятся к компонентам и переносятся в компонентный граф.

#### *11.1.3 Принципы объектно-ориентированного проектирования*

См 9.3

## **11.2 Шаблоны проектирования.**

Шаблоны проектирования — это допускающие многократное использование оптимизированные решения проблем программирования, с которыми мы сталкиваемся каждый день.

Некоторый шаблон, который должен быть реализован в надлежащей ситуации. Он не зависит от языка. Хороший шаблон проектирования должен быть таким, чтобы его можно было использовать с большинством языков (если не со всеми) в зависимости от характеристик языка. Чрезвычайно важно то, что любой шаблон проектирования необходимо использовать очень осторожно — если он применён в ненадлежащем месте, то его действие может быть разрушительным и породить много проблем для вас. Однако применённый в нужном месте в нужное время он может стать вашим спасителем.

Шаблоны проектирования в принципе являются хорошо продуманными решениями проблем программирования. Многие программисты уже сталкивались ранее с этими проблемами и использовали для преодоления эти «решения».

Есть три основных типа шаблонов проектирования:

- Структурные шаблоны, в общем случае, имеют дело с отношениями между объектами, облегчая их совместную работу.
- Порождающие шаблоны обеспечивают механизмы инстанцирования, облегчая создание объектов способом, который наиболее соответствует ситуации.
- Поведенческие шаблоны используются в коммуникации между объектами, делая её более лёгкой и гибкой.

### 11.2.1 Шаблон «Стратегия»

Используем его когда:

- Нужно вынести логику за пределы класса
- В рантайме менять алгоритмы

Шаблон «Стратегия» является поведенческим шаблоном проектирования, который позволяет вам решать, какой план действий должна принять программа, основываясь на определённом контексте при выполнении. Вы закладываете два различных алгоритма внутри двух классов и решаете в ходе выполнения, с какой стратегией следует работать.

Предположим, что вы в данный момент разрабатываете класс, который может или обновить или создать новую пользовательскую запись. Хотя ему требуются те же самые входы (имя, адрес, номер мобильного телефона и т.п.), но в зависимости от ситуации он должен использовать различные функции при обновлении и создании.

```
class User {

    public function CreateOrUpdate($name, $address, $mobile, $userid = null)
    {
        if( is_null($userid) ) {
            // Это значит, что пользователь ещё не существует, надо создать новую
            // запись
        } else {
            // Это значит, что пользователь уже существует, необходимо обновить на
            // зе данного идентификатора пользователя
        }
    }
}
```

### 11.2.2 Шаблон «Адаптер»

Адаптер (англ. Adapter) — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

Другими словами — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Система поддерживает требуемые данные и поведение, но имеет неподходящий интерфейс.

Адаптер предусматривает создание класса-оболочки[1] с требуемым интерфейсом.

### *11.2.3 Шаблон «объектный пул»*

Объектный пул (англ. object pool) — порождающий шаблон проектирования, набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создаётся, а берётся из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул.

Я так сокеты делал для TCP клиентов

### *11.2.4 Шаблон «Итератор»*

Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящих в состав агрегации.

### *11.2.5 Шаблон «Фабричный метод»*

Порождающий шаблон проектирования, определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инициализировать.

### *11.2.6 Шаблон «Декоратор»*

Декоратор (англ. Decorator) — структурный шаблон проектирования, расширяющий функциональность другого класса без использования наследования.

Класс, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

### *11.2.7 Шаблон «Одиночка»*

Класс, который может иметь только один экземпляр.

### 11.2.8 Шаблон «Делегирование»

Объект внешне выражает некоторое поведение, но в реальности передаёт ответственность за выполнение этого поведения связанному объекту.

## 11.3 Моделирование программных систем, язык UML.

UML (англ. Unified Modeling Language — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения, системного проектирования и отображения организационных структур.

UML является языком широкого профиля, это — открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

**Композиция** — один класс не может существовать без другого.

```
class Human
{
public:
    void Think()
    {
        brain.Think();
    }

private:
    class Brain {
public:
    void Think()
    {
        cout << "Я думаю!" << endl;
    }
};

    Brain brain;
};
```

**Агрегация** — когда один объект (контейнер) имеет ссылку на другой объект. Оба объекта могут существовать независимо: если контейнер будет уничтожен, то его содержимое — нет. Как в пред примере есть кепка, которая описана отдельно от человека, и может существовать без человека.

....

## Типы диаграмм

### Структурные

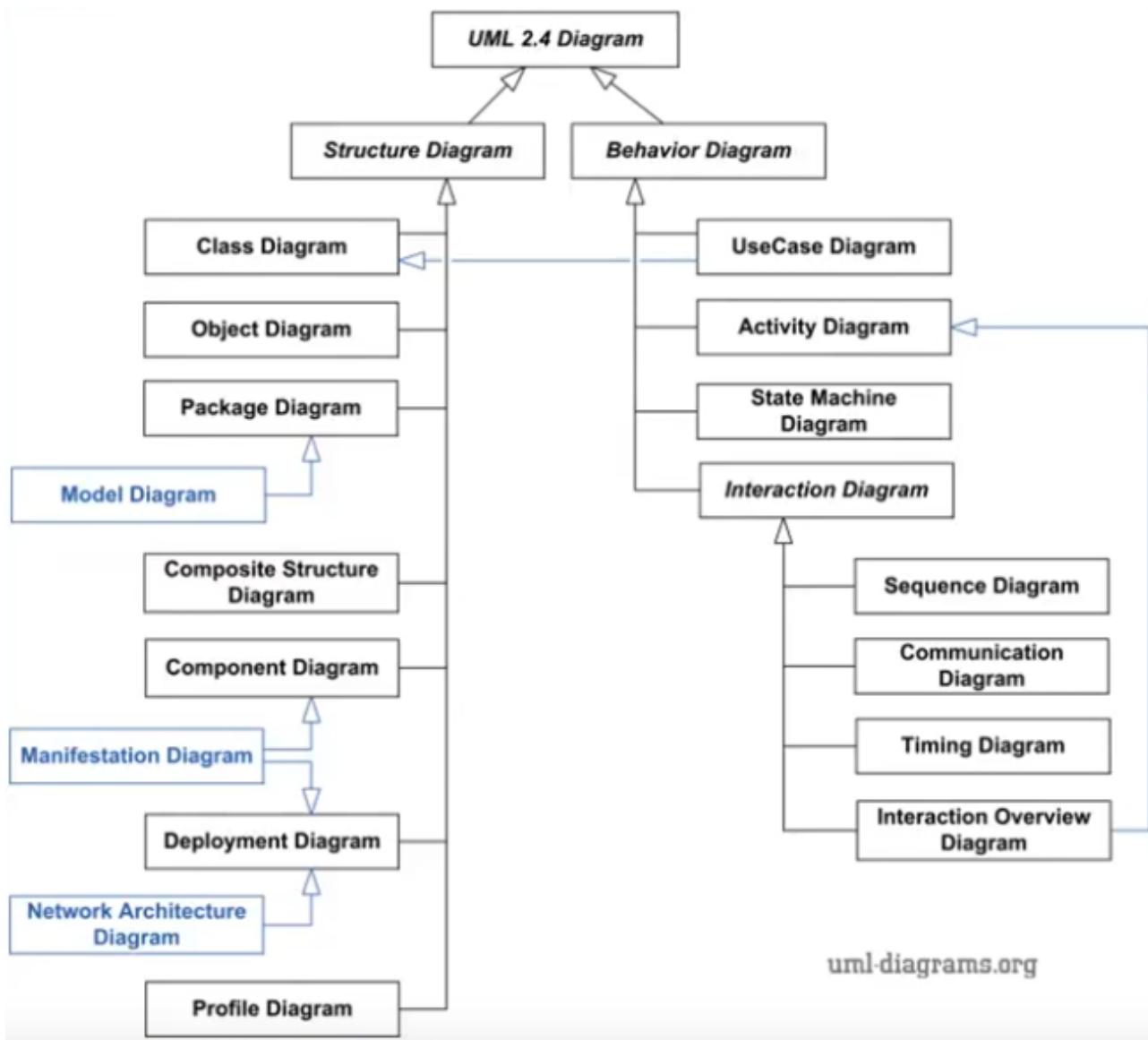
structure diagrams

- Общая картина взаимодействия, статичная структура.
- Как все устроено, кто с кем связан

### Поведенческие

behavior diagrams

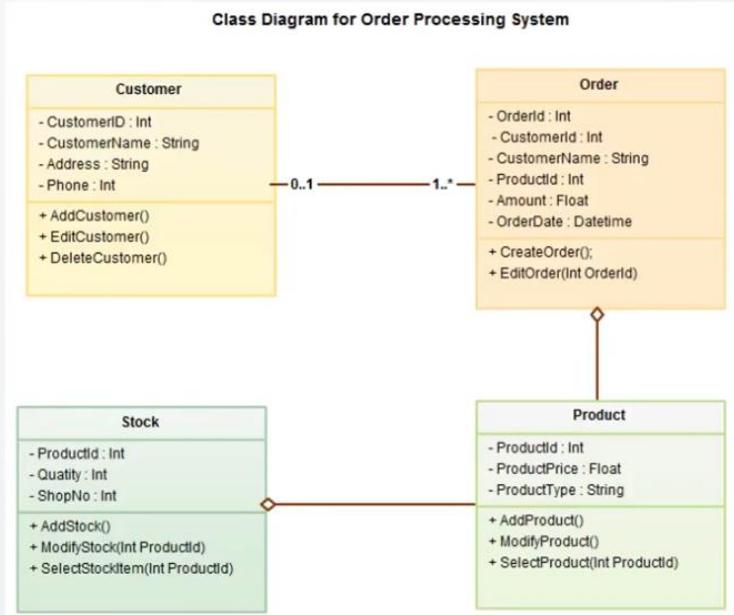
- Динамическое поведение, изменение состояния во времени
- Как это работает, последовательность процессов



### 11.3.1 Class

#### Class diagram

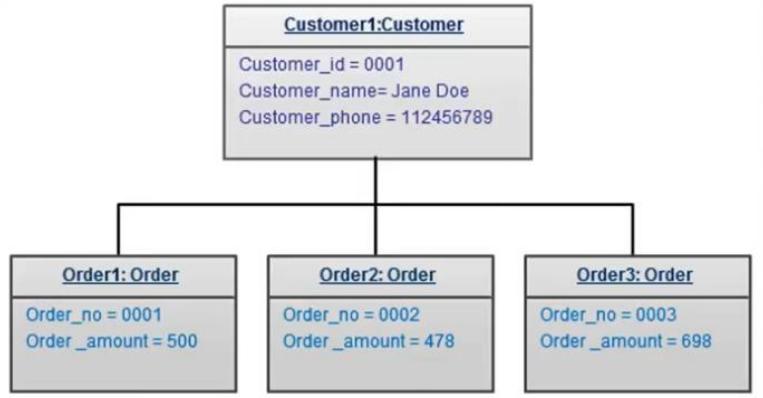
- Описание классов, интерфейсов, связей, полей методов
- Структура в стиле ООП
- Позволяет понять работу кода без изучения самого кода
- Часто сначала рисуются диаграммы, из которых автоматически генерится Java код



### 11.3.2 Object

#### Object diagram

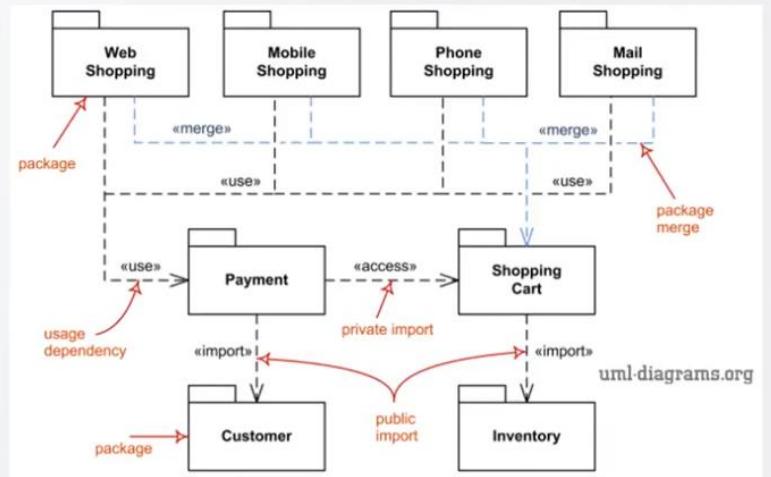
- Состояние экземпляров классов с конкретными значениями полей в определенный момент времени
- Похож на диаграмму классов, но в режиме **runtime** (во время работы программы)



### 11.3.3 Package

#### Package diagram

- Показывает вложенность и связи между пакетами
- Более высокий уровень, чем классы



### 11.3.4 Model

#### Model diagram

- Описание «слоев» проекта – из каких глобальных уровней состоит проект
- Используется для многоуровневых приложений
- Часто используется в ТЗ для общего описание частей проекта

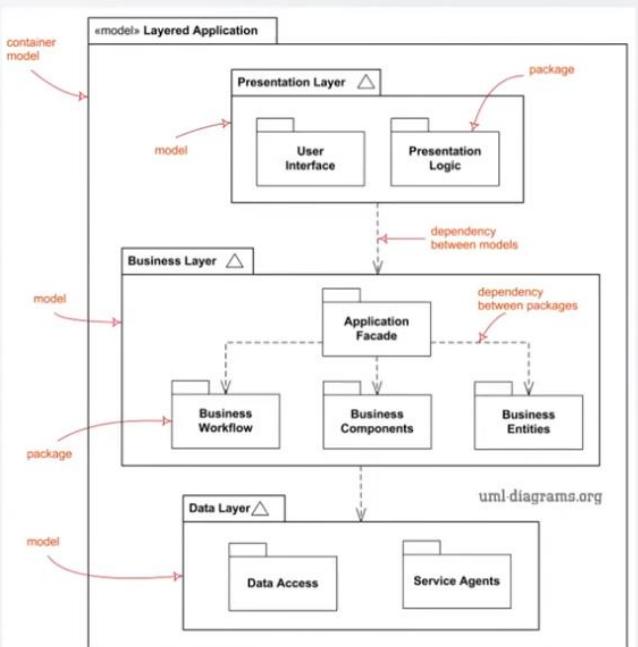
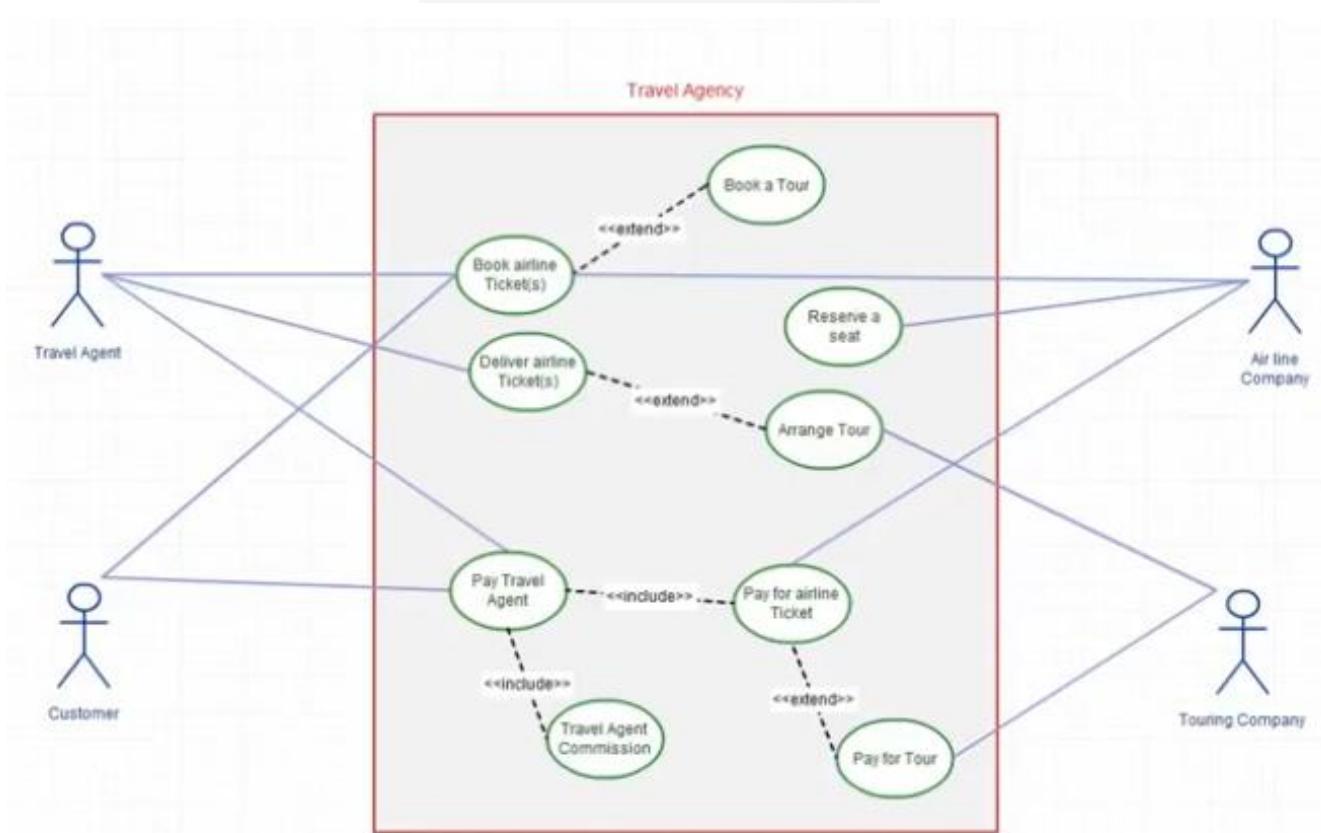


Рисунок 7 Можно описать MVC

### 11.3.5 UseCase

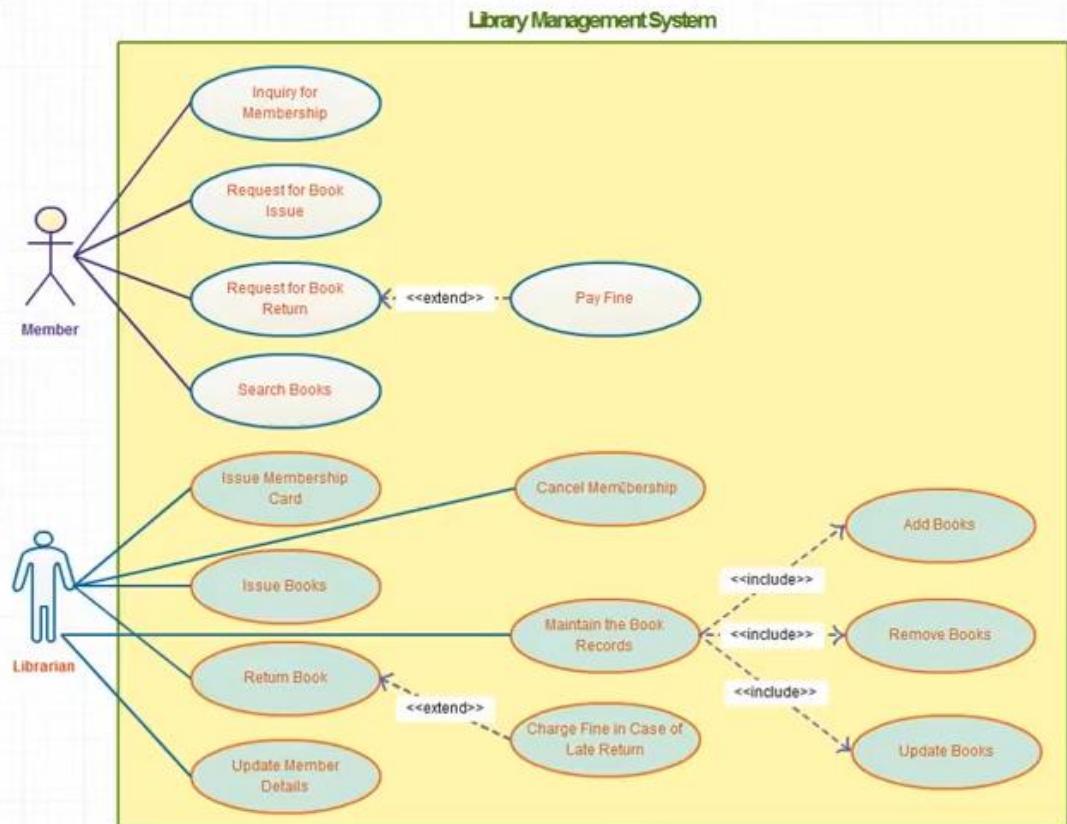
## UseCase diagram

- Диаграмма прецедентов, диаграмма вариантов использования
  - Описание возможных сценариев работы с системой с точки зрения пользователей
  - Возможные пути использования пользователем системы
  - Описание всех участников системы (актеры)



**UseCase diagram** - диаграмма прецедентов, диаграмма вариантов использования:

- Верхний уровень описание системы
- Описание возможных сценариев работы с системой с точки зрения пользователей
- Представление системы с точки зрения пользователя (точки взаимодействия с системой)
- Прецеденты помогают в разработке GUI
- Прецеденты не описывают, что пользователь может сделать с графическими компонентами (растягивать окно, нажимать на кнопки сортировки, закрывать и пр.) – это все детали реализации
- Разделяют инициатора (кто запускает процесс) и исполнителя (кто реализовывает)
- Инициатором может быть не только человек, но и другая система, таймер и пр.
- После описания прецедентов вы точно будете понимать, как можно будет работать с вашей системой



## Основные понятия

- **Actor** – пользователь, действующее лицо (инициатор или исполнитель цели)
- **UseCase** – прецедент, цель, которую хочет достигнуть пользователь
- **Association** – ассоциация, связь между элементами (актером и целями)
- **Include** – включение; возможные варианты реализации цели (или набор целей для реализации общей цели)
- **Extend** – расширение; новая цель, которая расширяет существующую (похоже на наследование класса в Java)

### 11.3.6 Activity

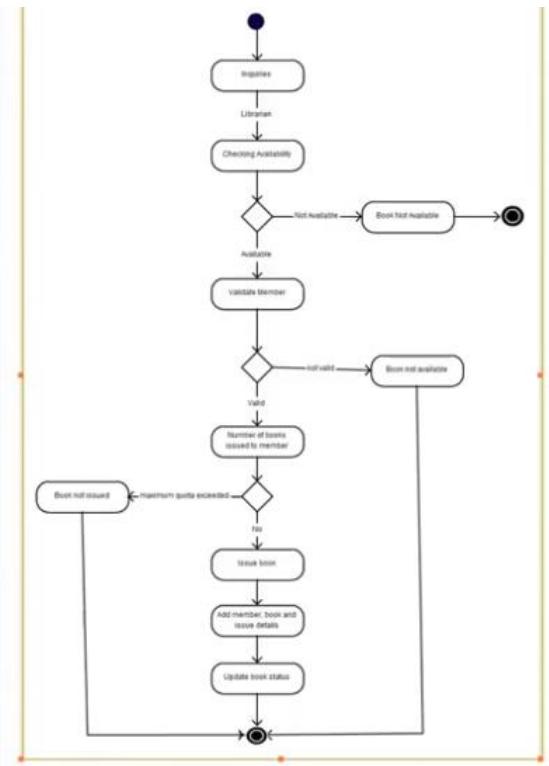
## Назначение

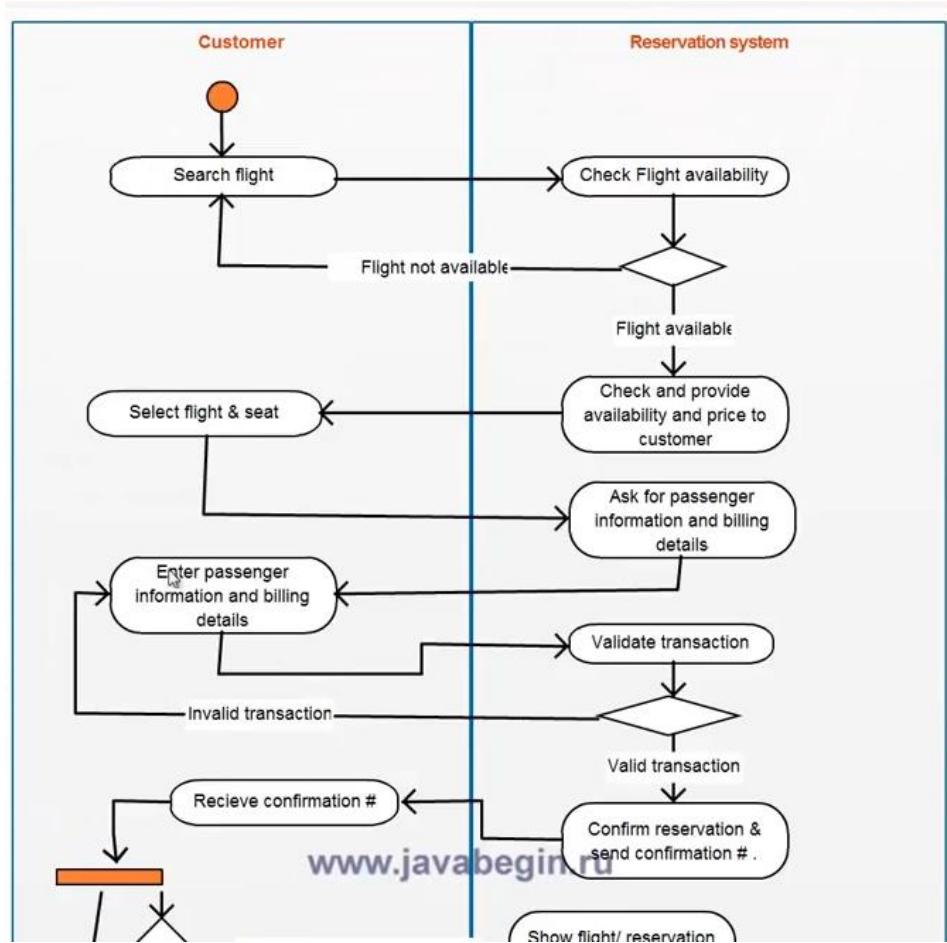
Activity diagram – диаграмма деятельности (активности):

- Описание одного процесса (алгоритма, активности, бизнес-процесса)
- Похожа на классическую блок-схему, но с некоторыми отличиями, дополнениями
- Показывает процесс в динамике, ход (текущее) работы в зависимости от условий
- Легче разрабатывать, если уже создана UseCase диаграмма (прецеденты)
- Изображается в терминах действий с условиями (также имеется возможность указания объектов)

## Activity diagram

- Описание возможных бизнес процессов приложения
- Взаимодействие «потоков», пошаговое представление действий
- Более низкий уровень, чем UseCase диаграмма
- Может быть конкретным описанием блоков UseCase диаграмм

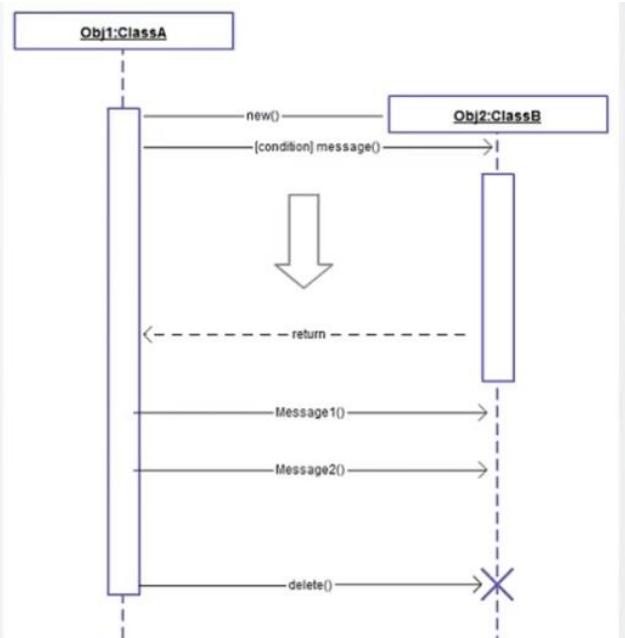


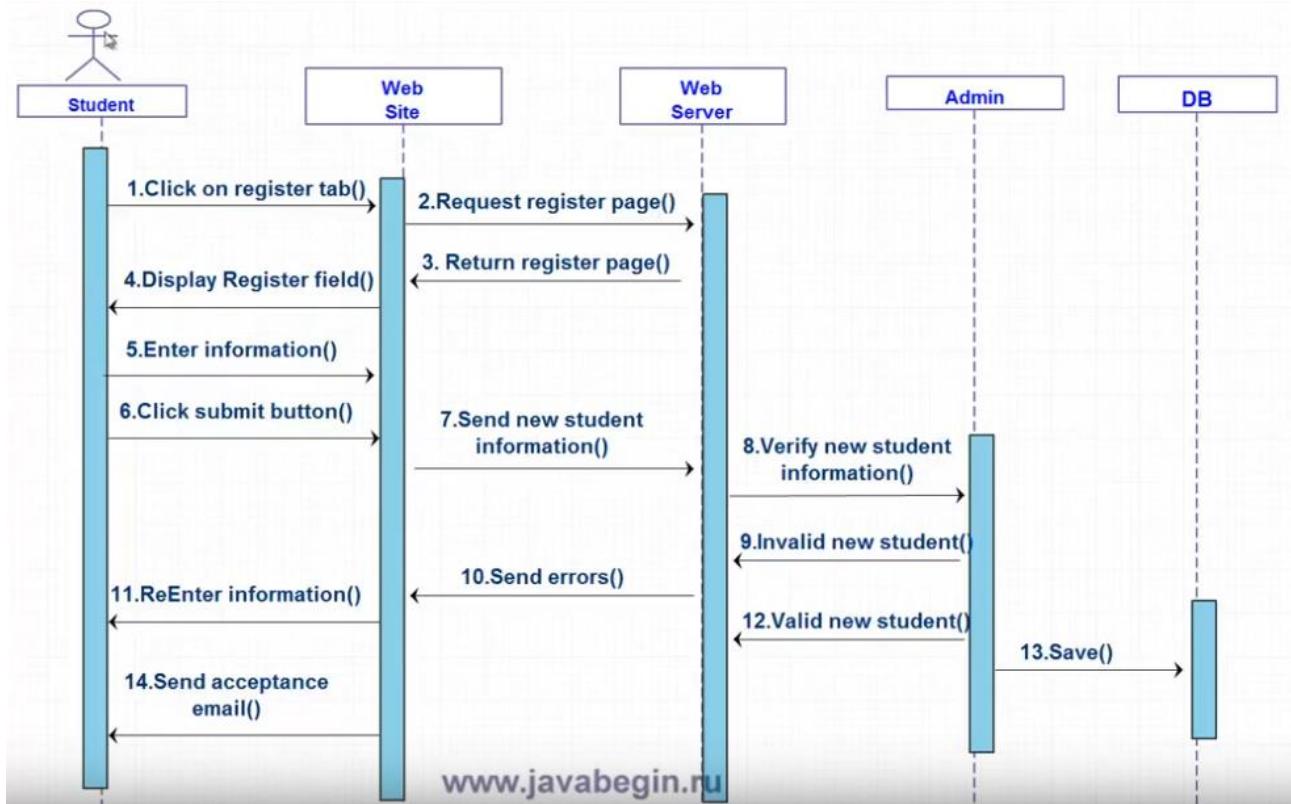


### 11.3.7 Sequence

## Sequence diagram

- Последовательность взаимодействия объектов для определенного бизнес процесса
- Как объекты друг друга вызывают и какие параметры передают
- В отличии от Class diagram или Object diagram - показывает объекты «в действии»
- Показывает время жизни объектов (создание, удаление)

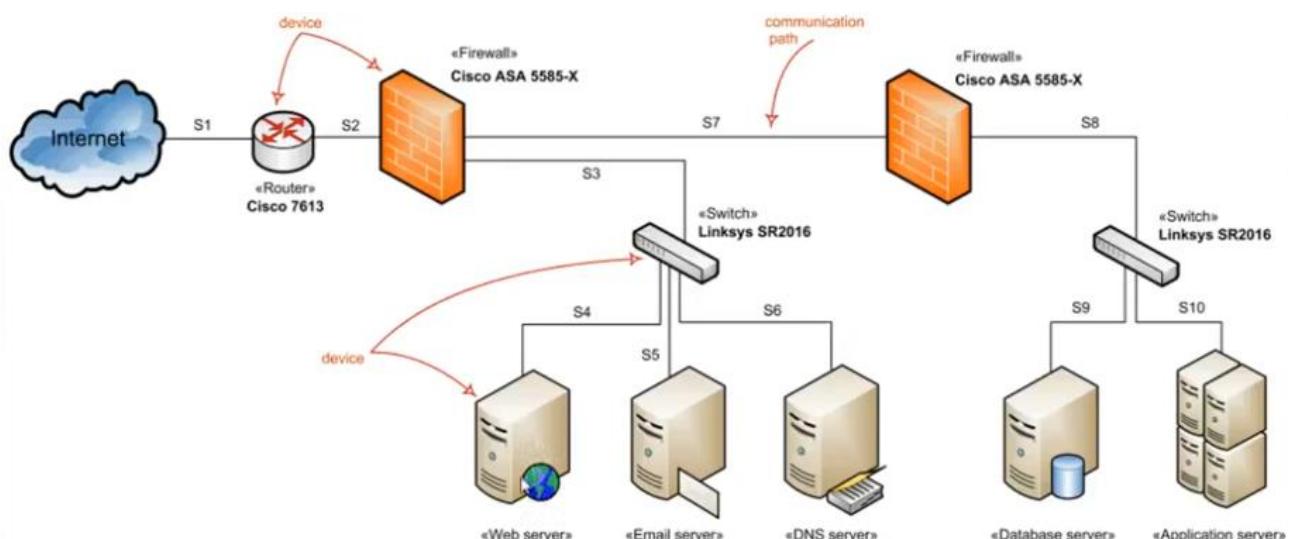




Sequence diagram – диаграмма последовательности (взаимодействия):

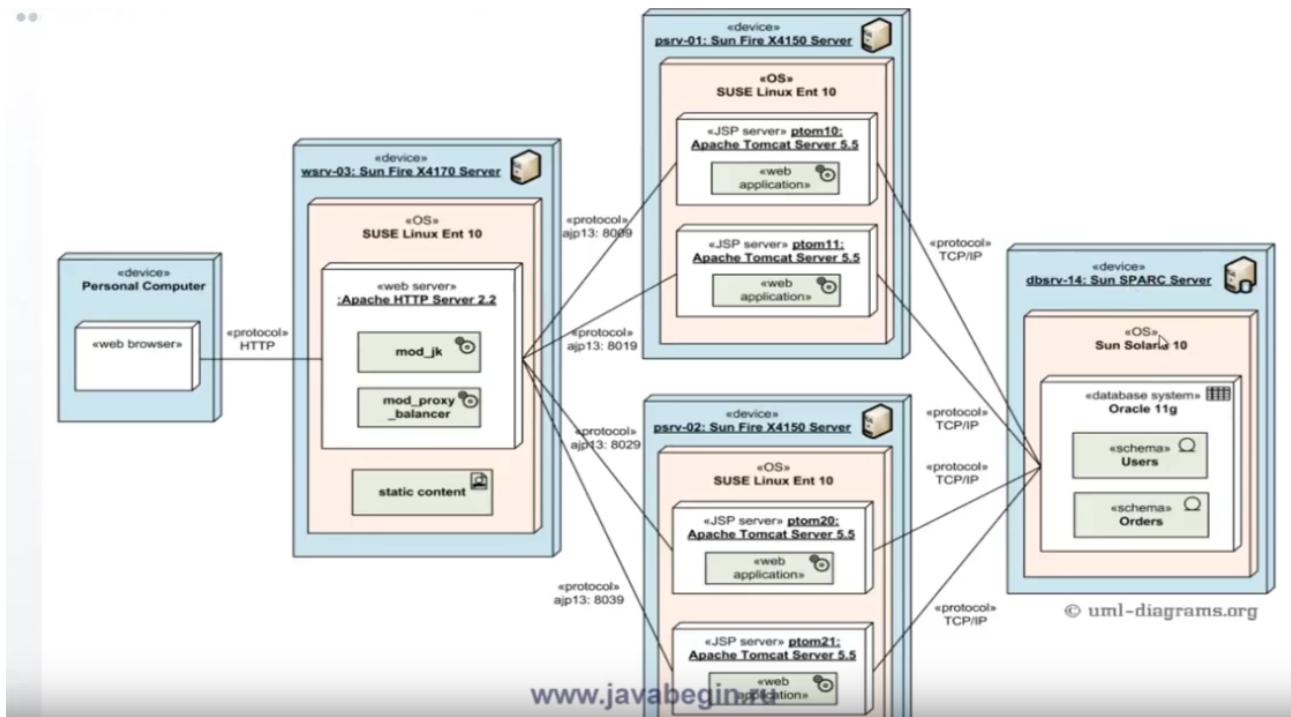
- Последовательность вызовов между объектами
- Обмен сообщениями между объектами
- Временная шкала всех действующих объектов (время жизни объекта)
- Более детальный обзор взаимодействия объектов (по сравнению с Activity diagram)
- Низкий уровень, который будет больше интересен разработчикам
- Используется шкала «слева-направо» и «сверху-вниз»

### 11.3.8 Deployment



## Deployment diagram – диаграмма развертывания:

- Описание архитектуры, топологии системы (ОС, БД, сервера и пр.)
- Информация для администраторов
- Ключевое понятие – «узел»



## 11.4 Современные подходы к автоматическому синтезу программ.

Комментарий от преподавателя ФИТ:

На текущий момент накоплен большой багаж в плане алгоритмов, моделей и пр. Многие задачи (математические, технические и пр.) могут быть решены с использованием стандартных алгоритмов. Алгоритмы же в свою очередь могут быть так или иначе оптимизированы под целевую платформу (процессор, например, или вычислительный комплекс). Существуют системы, которые по заданным параметрам создают оптимизированный код для таких систем.

Другой аспект, например, по формальному описанию интерфейса создание кода для взаимодействия клиентской или серверной части. Здесь можно посмотреть на Domain specific languages и разработку соответствующих языков.

Предметно-ориентированный язык (англ. domain-specific language, DSL — «язык, специфический для предметной области») — компьютерный язык, специализированный для конкретной области применения (в противоположность языку общего назначения, применимому к широкому

спектру областей и не учитывающему особенности конкретных сфер знаний). Построение такого языка и/или его структура данных отражают специфику решаемых с его помощью задач

Примеры DSL языков:

- LaTeX - для подготовки (компьютерной вёрстки) текстовых документов;
- SQL для СУБД;
- HTML для разметки документов;
- Verilog и VHDL для описания аппаратного обеспечения;
- XML, .ini, .conf.

Строго говоря, деление языков программирования на языки общего назначения и предметно-ориентированные весьма условно, особенно, если учесть, что формально любой протокол или формат файлов является языком.

В тему включаются такие вопросы, как грамматика, верификация программ и многое другое.

## 12

### **12.1 Современные методы и технологии построения распределённых программных систем**

Распределенная система - система, в которой обработка информации сосредоточена не на одной вычислительной машине, а распределена между несколькими компьютерами. При проектировании распределенных систем, которое имеет много общего с проектированием ПО в общем, все же следует учитывать некоторые специфические особенности.

Существует шесть основных характеристик распределенных систем.

- Совместное использование ресурсов. Распределенные системы допускают совместное использование как аппаратных (жестких дисков, принтеров), так и программных (файлов, компиляторов) ресурсов.
- Открытость. Это возможность расширения системы путем добавления новых ресурсов.
- Параллельность. В распределенных системах несколько процессов могут одновременно выполняться на разных компьютерах в сети. Эти процессы могут взаимодействовать во время их выполнения.

- Масштабируемость. Под масштабируемостью понимается возможность добавления новых свойств и методов.
- Отказоустойчивость. Наличие нескольких компьютеров позволяет дублирование информации и устойчивость к некоторым аппаратным и программным ошибкам. Распределенные системы в случае ошибки могут поддерживать частичную функциональность. Полный сбой в работе системы происходит только при сетевых ошибках.
- Прозрачность. Пользователям предоставляется полный доступ к ресурсам в системе, в то же время от них скрыта информация о распределении ресурсов по системе.

Задача разработчиков распределенных систем - спроектировать программное и аппаратное обеспечение так, чтобы предоставить все необходимые характеристики распределенной системы. А для этого требуется знать преимущества и недостатки различных архитектур распределенных систем. Выделяется три типа архитектур распределенных систем.

- Архитектура клиент/сервер. В этой модели систему можно представить как набор сервисов, предоставляемых серверами клиентам. В таких системах серверы и клиенты значительно отличаются друг от друга.
- Трехзвенная архитектура. В этой модели сервер предоставляет клиентам сервисы не напрямую, а посредством сервера бизнес-логики.
- Архитектура распределенных объектов. В этом случае между серверами и клиентами нет различий и систему можно представить как набор взаимодействующих объектов, местоположение которых не имеет особого значения. Между поставщиком сервисов и их пользователями не существует различий.

Про первые две модели было сказано уже не раз, остановимся подробнее на третьей.

Эта архитектура широко применяется в настоящее время и носит также название архитектуры веб-сервисов. Веб-сервис - это приложение, доступное через Internet и предоставляющее некоторые услуги, форма которых не зависит от поставщика (так как используется универсальный формат данных - XML) и платформы функционирования. В данное время существует три различные технологии, поддерживающие концепцию распределенных объектных систем.

### *12.1.1 J2EE*

Jakarta EE (ранее — Java Platform, Enterprise Edition, сокр. Java EE, до версии 5.0 — Java 2 Enterprise Edition или J2EE). В 2018 Eclipse Foundation переименовала Java EE в Jakarta EE — набор спецификаций и соответствующей документации для языка Java, описывающей архитектуру серверной платформы для задач средних и крупных предприятий.

Спецификации детализированы настолько, чтобы обеспечить переносимость программ с одной реализации платформы на другую. Основная цель спецификаций — обеспечить масштабируемость приложений и целостность данных во время работы системы. Java EE во многом ориентирована на использование её через веб, как в интернете, так и в локальных сетях.

Основная идея, лежавшая в разработке технологии Enterprise JavaBeans (EJB) — создать такую инфраструктуру для компонент, чтобы они могли быть легко “вставляться» (plug in) и удаляться из серверов, тем самым увеличивая или снижая функциональность сервера.

EJB-спецификация определяет следующие цели:

- Облегчить разработчикам создание приложений, избавив их от необходимости реализовывать с нуля такие сервисы, как транзакции (transactions), нити (threads), загрузка (load balancing) и другие. Разработчики могут сконцентрироваться на описании логики своих приложений, оставляя заботы о хранении, передаче и безопасности данных на EJB-систему. При этом все равно имеется возможность самому контролировать и описывать порученные системе процессы.
- Описать основные структуры EJB-системы, описав при этом интерфейсы взаимодействия (contracts) между ее компонентами.
- EJB преследует цель стать стандартом для разработки клиент/сервер приложений на Java. Таким же образом, как исходные JavaBeans (Delphi, или другие) компоненты от различных производителей можно было составлять вместе с помощью соответствующих RAD-систем (см ниже), получая в результате работоспособные клиенты, таким же образом серверные компоненты EJB от различных производителей также могут быть использованы вместе. EJB-компоненты, будучи Java-классами, должны без сомнения работать на любом EJB-совместимом сервере даже без перекомпиляции, что практически нереально для других систем.

- EJB совместима с Java API, может взаимодействовать с другими (не обязательно Java) приложениями, а также совместима с CORBA.

**RAD** (от англ. *rapid application development* — быстрая разработка приложений) — концепция организации технологического процесса разработки программных продуктов, ориентированная на максимально быстрое получение качественного результата в условиях сильных ограничений по срокам и бюджету и нечётко определённых требований к продукту.

Разработчику, однако, не нужно самому реализовывать EJB-объект. Этот класс создается специальным кодогенератором, поставляемым вместе в EJB-контейнером. Как уже было сказано, EJB-объект (созданный с помощью сервисов контейнера) и EJB-компонент (созданная разработчиком), реализуют один и тот же интерфейс. В результате, когда приложение-клиент хочет вызвать метод у EJB-компонента, то сначала вызывается аналогичный (по имени) метод у EJB-объекта, что находится на стороне клиента, а тот, в свою очередь, связывается с удаленной EJB-компонентой и вызывает у нее этот метод (с теми же аргументами).

Существует два различных типа ``бинов".

- Session bean представляет собой EJB-компоненту, связанную с одним клиентом. ``Бины" этого типа, как правило, имеют ограниченный срок жизни (хотя это и не обязательно), и редко участвуют в транзакциях. В частности, они обычно не восстанавливаются после сбоя сервера. В качестве примера session bean можно взять ``бин", который живет в Web-сервере и динамически создает HTML-страницы клиенту, при этом следя за тем, какая именно страница загружена у клиента. Когда же пользователь покидает Web-узел, или по истечении некоторого времени, session bean уничтожается. Несмотря на то, что в процессе своей работы, session bean мог сохранять некоторую информацию в базе данных, его предназначение заключается все-таки не в отображении состояния или в работе с ``вечными объектами", а просто в выполнении некоторых функций на стороне сервера от имени одного клиента.
- Entity bean, наоборот, представляет собой компоненту, работающую с постоянной (*persistent*) информацией, хранящейся, например, в базе данных. Entity beans ассоциируются с элементами баз данных и могут быть доступны одновременно нескольким пользователям. Так как информация в базе данных является постоянной, то и entity beans живут

постоянно, ``выживая'', тем самым, после сбоев сервера (когда сервер восстанавливается после сбоя, он может восстановить ``бин'' из базы данных). Например, entity bean может представлять собой строку какой-нибудь таблицы из базы данных, или даже результат операции SELECT. В объектно-ориентированных базах данных, entity bean может представлять собой отдельный объект, со всеми его атрибутами и связями.

EJB имеет следующие положительные и отрицательные стороны:

#### *12.1.1.1 Достоинства*

- Быстрое и простое создание
- Java-оптимизация
- Кроссплатформенность
- Динамическая загрузка компонент-переходников
- Возможность передачи объектов по значению
- Встроенная безопасность

#### *12.1.1.2 Недостатки*

- Поддержка только одного языка - Java
- Трудность интегрирования с существующими приложениями
- Плохая масштабируемость
- Производительность
- Отсутствие международной стандартизации

#### *12.1.2 .NET Framework*

.NET Framework — программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду.

Хотя .NET является патентованной технологией корпорации Microsoft и официально рассчитана на работу под операционными системами семейства Microsoft Windows, существуют независимые проекты (прежде всего это Mono и Portable.NET), позволяющие запускать программы .NET на некоторых других операционных системах. В настоящее время .NET Framework получает развитие в виде .NET Core, изначально предполагающей кроссплатформенную разработку и эксплуатацию.

Программа для .NET Framework, написанная на любом поддерживаемом языке программирования, сначала переводится компилятором в единый для .NET промежуточный байт-код Common Intermediate Language (CIL).

Использование виртуальной машины предпочтительно, так как избавляет разработчиков от необходимости заботиться об особенностях аппаратной части. В случае использования виртуальной машины CLR встроенный в неё JIT-компилятор «на лету» (just in time) преобразует промежуточный байт-код в машинные коды нужного процессора. Современная технология динамической компиляции позволяет достичь высокого уровня быстродействия. Виртуальная машина CLR также сама заботится о базовой безопасности, управлении памятью и системе исключений, избавляя разработчика от части работы.

Одной из основных идей Microsoft .NET является совместимость программных частей, написанных на разных языках. Например, служба, написанная на C++ для Microsoft .NET, может обратиться к методу класса из библиотеки, написанной на Delphi; на C# можно написать класс, наследованный от класса, написанного на Visual Basic .NET.

Языки, поставляемые вместе с Microsoft Visual Studio:

- C#
- Visual Basic .NET
- JScript .NET
- C++/CLI — новая версия Managed C++
- Ruby
- Python (IronPython)
- Delphi (Oxygene)
- F# — член семейства языков программирования ML, включён в VS2010/VS2012/VS2015/VS2017
- J# — последний раз был включён в VS2005

Объектные классы .NET, доступные для всех поддерживаемых языков программирования, содержатся в библиотеке Framework Class Library (FCL). В FCL входят классы

#### *12.1.2.1 Windows Forms*

Windows Forms — интерфейс программирования приложений (API), отвечающий за графический интерфейс пользователя и являющийся частью Microsoft .NET Framework. Данный интерфейс упрощает доступ к элементам

интерфейса Microsoft Windows за счет создания обёртки для существующего Win32 API в управляемом коде. Причём управляемый код — классы, реализующие API для Windows Forms, не зависят от языка разработки. То есть программист одинаково может использовать Windows Forms как при написании ПО на C#, C++, так и на VB.Net, J# и др.

Windows Forms фактически является лишь оберткой Windows API-компонентов, и ряд её методов осуществляет прямой доступ к Win32-функциям обратного вызова (Callback), которые недоступны на других платформах.

С выходом .NET Framework 3.0 Microsoft выпустила новый API для рисования пользовательских интерфейсов: Windows Presentation Foundation, который базировался на DirectX 11 и декларативном языке описания интерфейсов XAML. Однако, даже несмотря на все это, Windows Forms и WPF всё ещё предлагают схожую функциональность, и поэтому Windows Forms не был упразднен в пользу WPF, а продолжает использоваться как альтернативная технология построения интерфейсов наряду с WPF.

### *12.1.2.2 WPF*

Windows Presentation Foundation (WPF) — аналог WinForms, система для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем, графическая (презентационная) подсистема в составе .NET Framework (начиная с версии 3.0), использующая язык XAML.

WPF предустановлена в Windows Vista (.NET Framework 3.0), Windows 7 (.NET Framework 3.5 SP1), Windows 8 (.NET Framework 4.0 и 4.5), Windows 8.1 (.NET Framework 4.5.1). С помощью WPF можно создавать широкий спектр как автономных, так и запускаемых в браузере приложений.

В основе WPF лежит векторная система визуализации, не зависящая от разрешения устройства вывода и созданная с учётом возможностей современного графического оборудования. WPF предоставляет средства для создания визуального интерфейса, включая язык XAML (eXtensible Application Markup Language), элементы управления, привязку данных, макеты, двухмерную и трёхмерную графику, анимацию, стили, шаблоны, документы, текст, мультимедиа и оформление[2].

Графической технологией, лежащей в основе WPF, является DirectX, в отличие от Windows Forms, где используется GDI/GDI+[3]. Производительность

WPF выше, чем у GDI+ за счёт использования аппаратного ускорения графики через DirectX.

Для работы с WPF требуется любой .NET-совместимый язык. В этот список входит множество языков: C#, F#, VB.NET, C++, Ruby, Python, Delphi (Prism), Lua и многие другие. Для полноценной работы может быть использована как Visual Studio, так и Expression Blend. Первая ориентирована на программирование, а вторая — на дизайн и позволяет делать многие вещи, не прибегая к ручному редактированию XAML. Примеры этому — анимация, стилизация, состояния, создание элементов управления и так далее.

### *12.1.2.3 ADO.NET*

ADO.NET (ActiveX Data Object для .NET) — технология, предоставляющая доступ и управление данными, хранящимся в базе данных или других источниках (Microsoft SQL Server, Microsoft Access, Microsoft Excel, Microsoft Outlook, Microsoft Exchange, Oracle, OLE DB, ODBC, XML, текстовые файлы), основанных на платформе .NET Framework и входящая в состав .NET Framework 2.0, представляет собой набор библиотек.

ADO.NET больше нацелена на автономную работу с помощью объектов DataSet. Объекты DataSet представляют локальные копии взаимосвязанных таблиц данных, каждая из которых содержит набор строк и столбцов. Объекты DataSet позволяют вызывающей сборке (наподобие веб-страницы или программы, выполняющейся на настольном компьютере) работать с содержимым DataSet, изменять его, не требуя подключения к источнику данных, и отправлять обратно блоки измененных данных для обработки с помощью соответствующего адаптера данных.

Технология ADO.NET построена так, чтобы изолировать программиста от изучения структур баз данных разных производителей, представляя поставщиков баз данных (data provider), которые инкапсулируют механизм работы с конкретной СУБД, что позволяет создавать адаптеры для любой СУБД и полностью использовать её особенности. Сделана такая абстракция для того, чтобы использовать одинаковые типы данных для работы с различными источниками данных, иметь общий подход (универсализацию) для работы с базами данных разных производителей, чтобы технология ADO.NET поддерживалась CLR

#### *12.1.2.4 ASP.NET*

ASP.NET (Active Server Pages для .NET) — платформа разработки веб-приложений, в состав которой входит: веб-сервисы, программная инфраструктура, модель программирования, от компании Майкрософт.

Поскольку ASP.NET основывается на Common Language Runtime (CLR), которая является основой всех приложений Microsoft .NET, разработчики могут писать код для ASP.NET, используя языки программирования, входящие в комплект .NET Framework (C#, Visual Basic.NET, J# и JScript .NET)[1].

Программная модель ASP.NET основывается на протоколе HTTP и использует его правила взаимодействия между сервером и браузером. При формировании страницы заложена абстрактная программная модель Web Forms и на ней основана основная часть реализации программного кода

#### *12.1.3 Web-сервисы*

Для начала несколько слов о том, что такое XML вообще. XML - универсальный формат данных, который используется для предоставления Web-сервисов. В основе Web-сервисов лежат открытые стандарты и протоколы: SOAP, UDDI и WSDL.

- SOAP (Simple Object Access Protocol), разработанный консорциумом W3C, определяет формат запросов к Web-сервисам. Сообщения между Web-сервисом и его пользователем пакуются в так называемые SOAP-конверты (SOAP envelopes, иногда их ещё называют XML-конвертами). Само сообщение может содержать либо запрос на осуществление какого-либо действия, либо ответ - результат выполнения этого действия.
- WSDL (Web Service Description Language). Интерфейс Web-сервиса описывается в WSDL-документах (а WSDL - это подмножество XML). Перед развертыванием службы разработчик составляет ее описание на языке WSDL, указывает адрес Web-сервиса, поддерживаемые протоколы, перечень допустимых операций, форматы запросов и ответов.
- UDDI (Universal Description, Discovery and Integration) - протокол поиска Web-сервисов в Internet (<http://www.uddi.org/>). Представляет собой бизнес-реестр, в котором провайдеры Web-сервисов регистрируют службы, а разработчики находят необходимые сервисы для включения в свои приложения.

#### *12.1.4 CORBA*

В конце 1980-х и начале 1990-х годов многие ведущие фирмы-разработчики были заняты поиском технологий, которые принесли бы ощутимую пользу на все более изменчивом рынке компьютерных разработок. В качестве такой технологии была определена область распределенных компьютерных систем. Необходимо было разработать единообразную архитектуру, которая позволяла бы осуществлять повторное использование и интеграцию кода, что было особенно важно для разработчиков. Цена за повторное использование кода и интеграцию кода была высока, но ни кто из разработчиков в одиночку не мог воплотить в реальность мечту о широко используемом, языково-независимом стандарте, включающем в себя поддержку сложных многосвязных приложений. Поэтому в мае 1989 была сформирована OMG (Object Management Group). Как уже отмечалось, сегодня OMG насчитывает более 700 членов (в OMG входят практически все крупнейшие производители ПО, за исключением Microsoft).

Задачей консорциума OMG является определение набора спецификаций, позволяющих строить интероперабельные информационные системы. Спецификация OMG -- The Common Object Request Broker Architecture (CORBA) является индустриальным стандартом, описывающим высокоуровневые средства поддерживания взаимодействия объектов в распределенных гетерогенных средах.

CORBA специфицирует инфраструктуру взаимодействия компонент (объектов) на представительском уровне и уровне приложений модели OSI. Она позволяет рассматривать все приложения в распределенной системе как объекты. Причем объекты могут одновременно играть роль и клиента, и сервера: роль клиента, если объект является инициатором вызова метода у другого объекта; роль сервера, если другой объект вызывает на нем какой-нибудь метод.

## **13**

### **13.1 Концепция типа и моделей данных**

Модель данных - это логическое представление данных и совокупность операций над ними, у нее есть три аспекта:

- аспект структуры: методы описания типов и логических структур данных в базе данных; что из себя логически представляет база данных;

- аспект манипуляции: методы манипулирования данными; определяет способы перехода между состояниями базы данных (то есть способы модификации данных) и способы извлечения данных из базы данных;
- аспект целостности: методы описания и поддержки целостности базы данных; определяет средства описаний корректных состояний базы данных.

С помощью модели данных могут быть представленные объекты предметной области, взаимосвязи между ними.

Современная СУБД базируется на использовании следующий моделях:

## **Основные концепции обработки данных**

### **Концепция файловой системы**

- данные ИС представляются в виде совокупности файлов несущей Операционной системы;
- структура файла определяется разработчиком ИС (совмещение логической и физической структуры данных);
- любая программа, содержащая запрос к данным, должна уметь интерпретировать структуру необходимых для реализации запроса файлов (зависимость программ обработки от организации данных);
- каждый запрос требует в принципе своей программы обработки (разнообразие точек зрения алгоритмическую обработку данных).

### **Концепция баз данных**

- данные ИС размещаются в файлах несущей Операционной системы;
- физическая структура файлов фиксируется, а логическая структура данных представляется на ее основе (разделение и физической и логической структуры данных);
- обработка запроса к данным реализуется через специальный интерфейс манипулирования данными (независимость программ от организации данных);
- программы разрабатываются на основе единой точки зрения на процедуры обработки данных.

### **Концепция объектно-ориентированных баз данных**

- данные ИС размещаются в файлах несущей Операционной системы;
- информация ИС представляется в виде объектов (логическая структура и методы - принцип разделения логической и физической структуры остается в силе);
- обработка запроса к данным реализуется через интерфейс манипулирования объектами (независимость программ и объектов);
- программы разрабатываются на основе единой точки зрения на классы объектов и идеи повторного использования программного кода.

## Методы обработки данных

### Позадачный метод

*Суть:* декомпозиция программы на подзадачи о своими блоками данных, своими алгоритмами.

*Проблемы:*

- избыточность данных;
- взаимосвязь между данными и программой;
- замкнутость системы;
- нарушаются естественная для учреждения схема обработки данных.

### Метод баз данных

*Суть:* наличие отдельного описания логической структуры данных, единая точка зрения на процедуры обработки данных.

*Проблемы:*

- проблема представления данных;
- проблема обучения персонала;
- структурная реорганизация учреждения (реинжениринг).

### Метод объектно-ориентированных баз данных

*Суть:* информация (данные + программы) представляется объектами, которые обмениваются сообщениями (взаимосвязь процессов).

*Проблемы:*

- проблема определения объектов (классы);
- проблема обучения персонала;
- реорганизация учреждения.

#### 13.1.1 ER-модель / сущность-связь

ER-модель (от англ. entity-relationship model, модель «сущность — связь») — модель данных, позволяющая описывать концептуальные схемы предметной области.

ER-модель используется при высокоуровневом (концептуальном) проектировании баз данных. С её помощью можно выделить ключевые сущности и обозначить связи, которые могут устанавливаться между этими сущностями.

Отметим, что модель "сущность-связь" не является моделью данных, поскольку не определяет операций над данными и ограничивается описанием только их логической структуры.

Любой фрагмент предметной области может быть представлен как множество сущностей, между которыми существует некоторое множество связей. Дадим определения:

**Сущность** (entity) - это объект, который может быть идентифицирован неким способом, отличающим его от других объектов. Примеры: конкретный человек, предприятие, событие и т.д.

**Набор сущностей** (entity set) - множество сущностей одного типа (обладающих одинаковыми свойствами). Примеры: все люди, предприятия, праздники и т.д. Наборы сущностей не обязательно должны быть непересекающимися. Например, сущность, принадлежащая к набору МУЖЧИНЫ, также принадлежит набору ЛЮДИ.

**Связь** (relationship) - это ассоциация, установленная между несколькими сущностями. Иерархическая модель данных.

Примеры:

- поскольку каждый сотрудник работает в каком-либо отделе, между сущностями СОТРУДНИК и ОТДЕЛ существует связь "работает в" или ОТДЕЛ-РАБОТНИК;
- так как один из работников отдела является его руководителем, то между сущностями СОТРУДНИК и ОТДЕЛ имеется связь "руководит" или ОТДЕЛ-РУКОВОДИТЕЛЬ;
- могут существовать и связи между сущностями одного типа, например связь РОДИТЕЛЬ - ПОТОМОК между двумя сущностями ЧЕЛОВЕК;

Роль сущности в связи - функция, которую выполняет сущность в данной связи. Например, в связи РОДИТЕЛЬ-ПОТОМОК сущности ЧЕЛОВЕК могут иметь роли "родитель" и "потомок". Указание ролей в модели "сущность-связь" не является обязательным и служит для уточнения семантики связи.

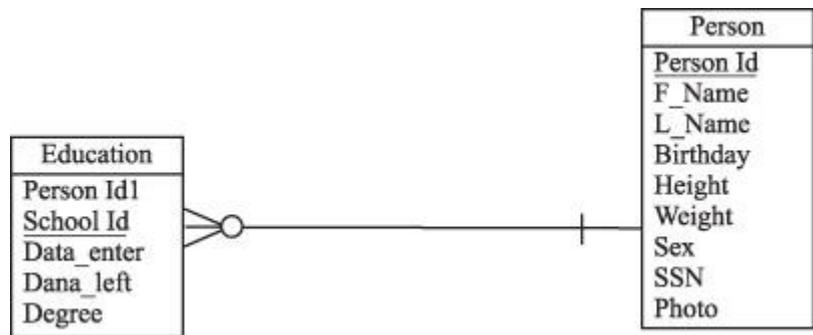
Набор связей (relationship set) - это отношение между n (причем n не меньше 2) сущностями, каждая из которых относится к некоторому набору сущностей.

Отношение (связь) сущностей на ER -диаграмме изображается линией, соединяющей эти сущности.

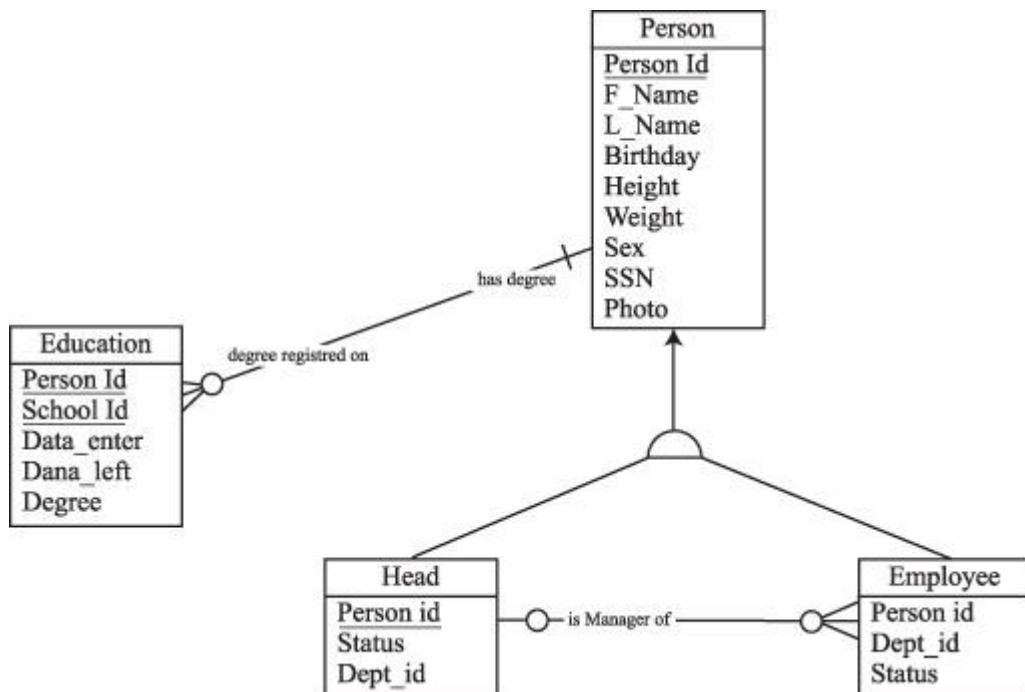
Степень связи изображается с помощью символа "птичья лапка" 1, указывающего на то, что в связи участвует много (N) экземпляров сущности, и одинарной горизонтальной чертой, указывающей на то, что в связи участвует один экземпляр сущности.

Необязательный класс принадлежности сущности к связи изображается с помощью кружочка на линии отношения рядом с сущностью, обязательный класс принадлежности - с помощью вертикальной черты на линии отношения рядом с сущностью.

Отношение читается вдоль линии либо слева направо, либо справа налево. На рис представлено следующее отношение: каждая специальность по образованию должна быть зарегистрирована за определенным физическим лицом (персоной), физическое лицо может иметь одну или более специальностей по образованию.



Супертипы и подтипы, так же как и сущности, обозначаются на ER-диаграмме с помощью прямоугольников. Отношения между ними изображаются с помощью "вилки", имеющей в точке ветвления полукруг.



Супертип Person (Персона) содержит общие для своих подтипов Head (Руководитель) и Employee (Служащий) атрибуты. Подтипы содержат только атрибуты, характерные для выделенных категорий. Предложенная конструкция

реализует отношение подчиненности в иерархии организации согласно ее штатному расписанию.

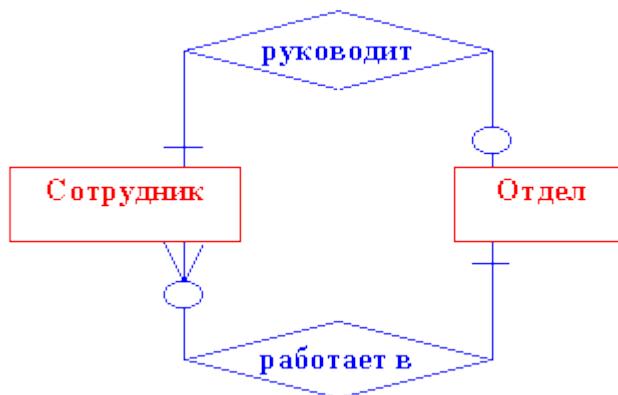
Изучив и проверив качество информационной модели данных предметной области, представленной в виде набора ER-диаграмм, проектировщик базы данных может приступать к созданию логической модели базы данных.

Могут существовать следующие степени бинарных связей:

#### 13.1.1.1 один ко многим ( $1 : n$ )

У одного человека много сотовых номеров, но каждый сотик указывает на конкретного человека.

В данном случае сущности с одной ролью может соответствовать любое число сущностей с другой ролью. Такова связь ОТДЕЛ-СОТРУДНИК. В каждом отделе может работать произвольное число сотрудников, но сотрудник может работать только в одном отделе. Графически степень связи  $n$  отображается "древообразной" линией, так это сделано на следующем рисунке.



Данный рисунок дополнительно иллюстрирует тот факт, что между двумя сущностями может быть определено несколько наборов связей.

Здесь также необходимо учитывать класс принадлежности сущностей. Каждый сотрудник должен работать в каком-либо отделе, но не каждый отдел (например, вновь сформированный) должен включать хотя бы одного сотрудника. Поэтому сущность "ОТДЕЛ" имеет обязательный, а сущность "СОТРУДНИК" необязательный классы принадлежности. Кардинальность бинарных связей степени  $n$  будем обозначать так:



$(0..N)$



$(1..N)$

### *13.1.1.2 много к одному ( $n : 1$ ).*

Эта связь аналогична отображению 1 : n. Предположим, что рассматриваемое нами предприятие строит свою деятельность на основании контрактов, заключаемых с заказчиками. Этот факт отображается в модели "сущность-связь" с помощью связи КОНТРАКТ-ЗАКАЗЧИК, объединяющей сущности КОНТРАКТ(НОМЕР, СРОК\_ИСПОЛНЕНИЯ, СУММА) и ЗАКАЗЧИК(НАИМЕНОВАНИЕ, АДРЕС). Так как с одним заказчиком может быть заключено более одного контракта, то связь КОНТРАКТ-ЗАКАЗЧИК между этими сущностями будет иметь степень n : 1.



В данном случае, по совершенно очевидным соображениям (каждый контракт заключен с конкретным заказчиком, а каждый заказчик имеет хотя бы один контракт, иначе он не был бы таковым), каждая сущность имеет обязательный класс принадлежности.

### *13.1.1.3 многие ко многим ( $n : n$ ).*

Есть книги и авторы. У одного автора может быть много книг, и у книги может быть много авторов. Строится отдельная таблица, в которой указана эта зависимость.

В этом случае каждая из ассоциированных сущностей может быть представлена любым количеством экземпляров. Пусть на рассматриваемом нами предприятии для выполнения каждого контракта создается рабочая группа, в которую входят сотрудники разных отделов. Поскольку каждый сотрудник может входить в несколько (в том числе и ни в одну) рабочих групп, а каждая группа должна включать не менее одного сотрудника, то связь между сущностями СОТРУДНИК и РАБОЧАЯ\_ГРУППА имеет степень n : n.



Если существование сущности x зависит от существования сущности y, то x называется зависимой сущностью (иногда сущность x называют "слабой", а "сущность" y - сильной). В качестве примера рассмотрим связь между ранее описанными сущностями РАБОЧАЯ\_ГРУППА и КОНТРАКТ. Рабочая группа создается только после того, как будет подписан контракт с заказчиком, и

прекращает свое существование по выполнению контракта. Таким образом, сущность РАБОЧАЯ\_ГРУППА является зависимой от сущности КОНТРАКТ. Зависимую сущность будем обозначать двойным прямоугольником, а ее связь с сильной сущностью линией со стрелкой:



Заметим, что кардинальность связи для сильной сущности всегда будет (1,1). Класс принадлежности и степень связи для зависимой сущности могут быть любыми. Предположим, например, что рассматриваемое нами предприятие пользуется некоторыми банковскими кредитами, которые представляются набором сущностей КРЕДИТ(НОМЕР\_ДОГОВОРА,СУММА, СРОК\_ПОГАШЕНИЯ, БАНК). По каждому кредиту должны осуществляться выплаты процентов и платежи в счет его погашения. Этот факт представляется набором сущностей ПЛАТЕЖ(ДАТА, СУММА) и набором связей "осуществляется по". В том случае, когда получение запланированного кредита отменяется, информация о нем должна быть удалена из базы данных. Соответственно, должны быть удалены и все сведения о плановых платежах по этому кредиту. Таким образом, сущность ПЛАТЕЖ зависит от сущности КРЕДИТ.



#### *13.1.1.4 Один к одному (1 : 1)*

Это означает, что в такой связи сущности с одной ролью всегда соответствует не более одной сущности с другой ролью. В рассмотренном нами примере это связь "руководит", поскольку в каждом отделе может быть только один начальник, а сотрудник может руководить только в одном отделе. Данный факт представлен на следующем рисунке, где прямоугольники обозначают сущности, а ромб - связь. Так как степень связи для каждой сущности равна 1, то они соединяются одной линией.

Чаще всего такое отношение является недоработкой бд и следует объединить таблицы в одну, но бывают ситуации, когда это необходимо.



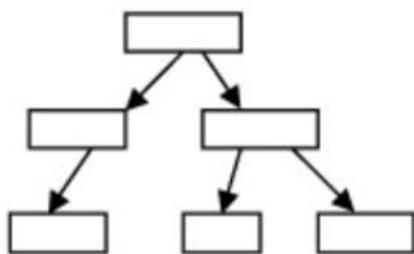
Другой важной характеристикой связи помимо ее степени является класс принадлежности входящих в нее сущностей или кардинальность связи. Так как в каждом отделе обязательно должен быть руководитель, то каждой сущности "ОТДЕЛ" непременно должна соответствовать сущность "СОТРУДНИК". Однако, не каждый сотрудник является руководителем отдела, следовательно в данной связи не каждая сущность "СОТРУДНИК" имеет ассоциированную с ней сущность "ОТДЕЛ".

Таким образом, говорят, что сущность "СОТРУДНИК" имеет обязательный класс принадлежности (этот факт обозначается также указанием интервала числа возможных вхождений сущности в связь, в данном случае это 1,1), а сущность "ОТДЕЛ" имеет необязательный класс принадлежности (0,1). Теперь данную связь мы можем описать как 0,1:1,1. В дальнейшем кардинальность бинарных связей степени 1 будем обозначать следующим образом:



### 13.1.2 Иерархическая модель данных

Иерархическая модель данных организует данные в виде древовидной. Примером простого иерархического представления может служить административная структура организации.



**Иерархическая**  
Взаимосвязи между данными жестко фиксированы.  
Изменение связи ведет к реорганизации структуры.  
Число связей ограничено.

Деревом в информатике называют совокупность корневого элемента и множества подчиненных ему элементов, в которой отношения между

элементами носят подчиненный вертикальный характер. Горизонтальные связи в такой системе отношений не допускаются.

Особенностью такого представления данных является наличие нескольких подчиненных уровней. В иерархической модели имеется корневой узел или корень дерева. Он располагается на 1-м, самом высоком уровне и не имеет узлов-предшественников. Остальные узлы называются порожденными и связаны между собой следующим образом: каждый узел имеет исходный, находящийся на вышестоящем уровне. На следующем уровне каждый узел может иметь более одного узла-потомка или не иметь потомков вовсе. Узлы, не имеющие порожденных, называются листьями. В иерархии рассматривают уровни, на которых расположен тот или иной узел или совокупность узлов.

Между исходным узлом и порожденными узлами по условию модели существует связь "один-ко-многим" (или "многие-к-одному").

Иерархия должна удовлетворять следующим условиям:

- Иерархия имеет исходный узел (корень), из которого строится дерево. Каждое дерево имеет только один корень.
- Узел имеет непустое множество атрибутов, которые описывают объект, моделируемый в данном узле.
- Порожденные узлы могут добавляться в дерево как в вертикальном, так и в горизонтальном направлении.
- Доступ к порожденным узлам возможен только через исходный узел, поэтому существует только один путь доступа к каждому узлу.
- ~~Возможно существование нескольких экземпляров каждого узла каждого уровня. При этом каждый экземпляр исходного узла начинает логическую запись.~~

К основным недостаткам иерархической модели можно отнести:

- сложность отображения связи "многие-к-многим"
- усложнение операции включения новых объектов и удаления устаревших объектов непосредственно в базе данных (в особенности обновление и удаление связей);
- неоднозначность представления данных о предметной области.

### **Пример.**

Пусть требуется построить иерархическую модель о преподавателях, студентах и дисциплинах, которые преподаватели преподают, а студенты изучают.

Предположим, что каждый преподаватель может читать несколько дисциплин, а каждый студент также может изучать несколько дисциплин.

Один из возможных вариантов построения иерархической модели может быть таким. Корневым узлом является студент (Номер студента, ФИО, Номер группы). Для каждого студента при данном представлении имеется экземпляр корневого узла. Преподаватель и дисциплина объединяются в один порожденный узел (Табельный номер преподавателя, ФИО, Ученое звание, Кафедра, Дисциплина).

При такой организации данных достаточно легко получать ответы на запросы типа "выдать информацию о сдаче экзаменов студентами по различным дисциплинам". Однако при ответе на вопрос, какие преподаватели принимают экзамены по ВТ, необходимо просмотреть все записи порожденных узлов для каждого корневого узла. Для этого вопроса более подходит модель, в которой корневым узлом является преподаватель (Табельный номер преподавателя, ФИО, ученое звание, кафедра), а порожденным является студент (номер студента, ФИО, номер группы, дисциплина, дата сдачи, оценка, зачет).

В первом варианте модели для каждого студента дублируется информация (в экземпляре порожденного узла) о преподавателях и дисциплине, а во втором - для каждого преподавателя о студентах. Отсюда возникают проблемы включения и удаления данных. Из принципа иерархии следует, что экземпляр порожденного узла не может существовать в отсутствие соответствующего ему экземпляра исходного узла. Таким образом, невозможно без привлечения дополнительных способов включить в базу данных информацию о преподавателях, которые не принимают экзамены (для первого варианта схемы). Чтобы соблюсти принцип иерархии, нужно сформировать, например, пустой исходный узел и породить еще проблему интерпретации нуль-значений.

При удалении исходного узла автоматически удаляются экземпляры порожденных узлов. Так, для второго варианта представления модели удаления сведения о преподавателе (уволился) удаляются все сведения о студентах, у него обучавшихся, а следовательно, теряется информация, необходимая для оценки качества обучения студентов.

Первопричиной этих проблем является то обстоятельство, что иерархическая модель не поддерживает отношение M:N.

При использовании иерархической модели актуальной является проблема отслеживания связей, хотя и в более простом варианте. К возможным недостаткам иерархической модели можно отнести вероятную асимметрию

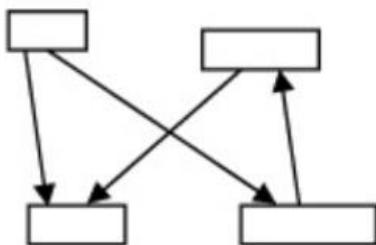
отношений между сущностями предметной области БД и неудобство в отображении горизонтальных связей, которые нужно выражать через вертикальные связи.

Пример: Information Management System фирмы IBM (1966-1968 г.).

### 13.1.3 Сетевая модель данных.

Сетевая модель данных — логическая модель данных, являющаяся расширением иерархического подхода.

Разница между иерархической моделью данных и сетевой состоит в том, что в иерархических структурах запись-потомок должна иметь в точности одного предка, а в сетевой структуре данных у потомка может иметься любое число предков.



**Сетевая**  
Характер связей более разнообразен.  
Трудно вводить изменения.

Достоинством сетевой модели данных является возможность эффективной реализации по показателям затрат памяти и оперативности.

Недостатком сетевой модели данных являются высокая сложность и жесткость схемы БД, построенной на её основе.

Пример самой первой СУБД – IDS 1960.

#### Пример

Рассмотрим отношение между следующими объектами: Студенческий коллектив, Студенческая группа, Комната в общежитии и Студент. Взаимосвязь между этими объектами не является иерархической, так как порожденный элемент Студент имеет два исходных - Студенческая группа и Комната в общежитии. Такие отношения, когда порожденный элемент имеет более одного исходного, описываются в виде сетевой структуры. В такой структуре любой элемент может быть связан с любым другим элементом.

Как и в случае иерархической модели, сетевую структуру можно описать в терминах исходных и порождаемых узлов, а также представить ее таким образом, чтобы порожденные узлы располагались ниже исходных. При

рассмотрении некоторых сетевых структур можно говорить об уровнях. Так, рассмотренная выше сетевая структура имеет три уровня.

Рассмотрим, как в сетевой модели будут представлены взаимосвязи между объектами. В нашем примере присутствуют два вида взаимосвязей: 1:М (Учебная группа - Студент) и М:1 (Студент - Комната в общежитии). Сетевые структуры, которые имеют такие связи между исходными и порожденными узлами, порожденными и исходными узлами, относят к простым сетевым структурам. Сложной сетевой структурой называют такую структуру, в которой присутствует хотя бы одна связь типа N:М. Примером такой связи является отношение Студент - Преподаватель. Такое разделение сетевых структур обусловлено технологическими сложностями реализации взаимосвязи N:М. Причем некоторые СУБД не обрабатывают сложных сетевых структур.

База данных с сетевой структурой состоит из нескольких областей. Каждая область состоит из записей, которые состоят из полей. Объединение записей в логическую структуру возможно не только по областям, но и с помощью наборов данных. По существу, набор данных - это поименованное двухуровневое дерево, которое является основой для построения многоуровневых деревьев.

Набор данных имеет следующие свойства:

- Набор данных есть поименованная совокупность связанных записей.
- В каждом экземпляре набора данных имеется только один экземпляр записи владельца.
- Экземпляр набора может содержать 0,1 или несколько записей-членов.
- Набор данных считается пустым, если ни один экземпляр записи-члена не связан с соответствующим экземпляром записи владельца.
- Экземпляр набора данных связан с записью владельца.
- Тип набора предполагает логическую взаимосвязь 1:М между владельцем и членом набора.
- Каждому типу набора данных присваивается имя, которое позволяет одной и той же паре типов объектов участвовать в нескольких взаимосвязях.

Необходимо различать тип и экземпляр набора. Предварительно поясним различие между понятиями "тип" и "экземпляр" записи.

Например, Студент является типом записи, а строка, содержащая информацию о конкретном студенте, является экземпляром типа записи Студент. Аналогичное различие существует между типом и экземпляром набора данных. Например, тип набора Состав группы, а его экземпляр содержит один

экземпляр типа записи владельца Учебная группа и N экземпляров типа записи-члена Студент. Определенный экземпляр типа записи-члена не может одновременно принадлежать более чем одному экземпляру типа записи-владельца. Уникальность владельца типа набора является обязательным элементом сетевой модели данных. С этой точки зрения иерархическая модель является частным случаем сетевой модели данных.

Сетевая модель данных является моделью объектов-связей, где допускаются только бинарные связи типа "многие-к-одному", что позволяет использовать для представления данных простую модель ориентированных графов. В некоторых определениях сетевой модели допускаются связи типа "многие-ко-многим", но требование бинарности связи остается в силе.

#### *13.1.4 Реляционная модель данных.*

См 15.1 стр 240

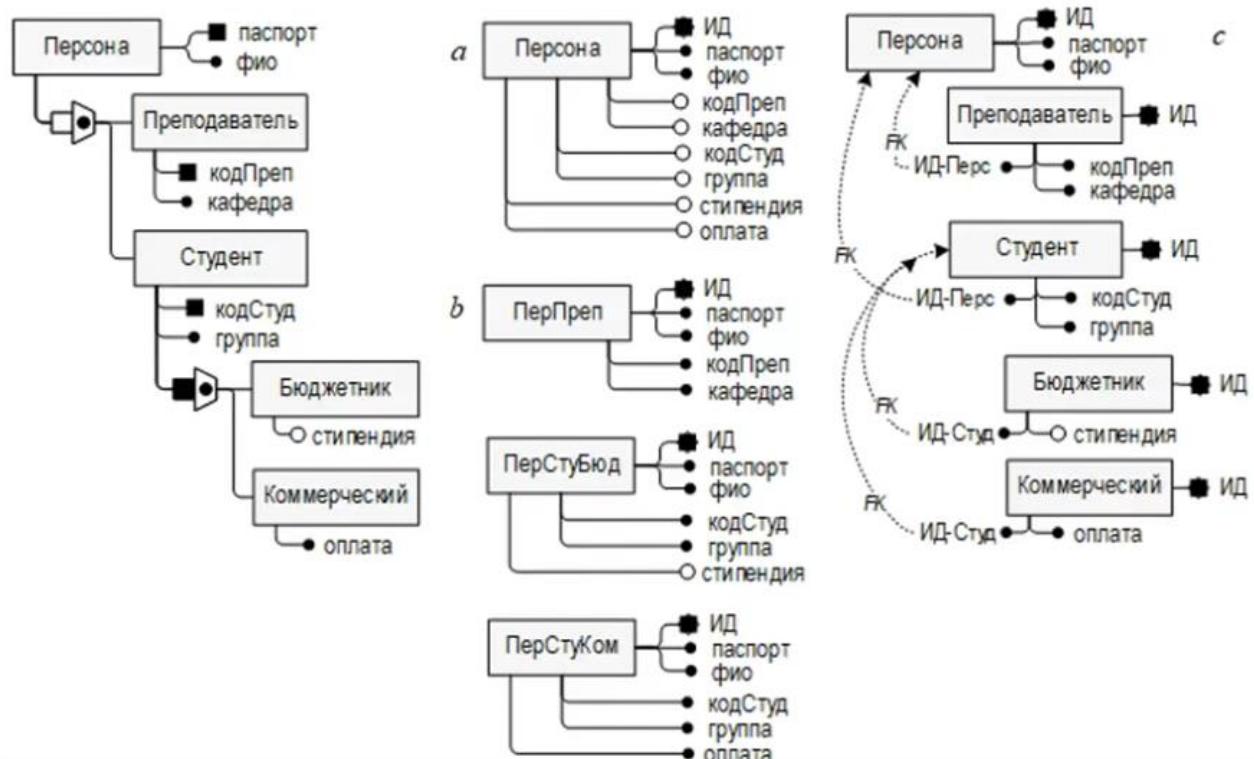
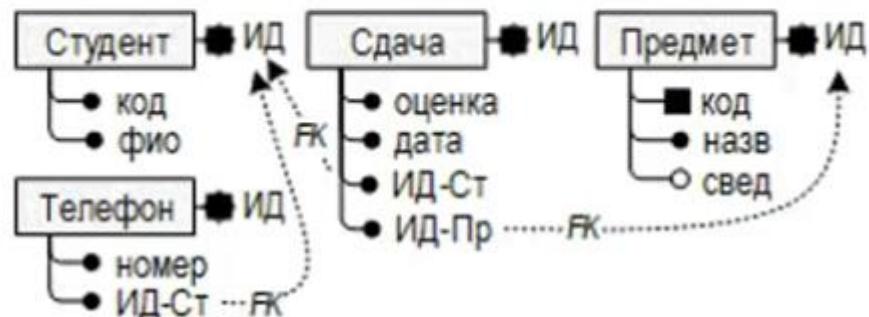
#### *13.1.5 Объектно-ориентированные БД*

Объектно-ориентированная база данных (ООБД) — база данных, в которой данные моделируются в виде объектов[1], их атрибутов, методов и классов[2].

Объектно-ориентированные базы данных обычно рекомендованы для тех случаев, когда требуется высокопроизводительная обработка данных, имеющих сложную структуру.

В манифесте ООБД[4] предлагаются обязательные характеристики, которым должна отвечать любая ООБД. Их выбор основан на 2 критериях:

- система должна быть объектно-ориентированной
- представлять собой базу данных.



Существует 2 реализации:

- действительно объектно-ориентированные субд (Cache, Db4o, ObjectStore)
- реляционные бд с интерфейсом ОО, Object-relational mapping (может быть на уровне клиента или сервера)

## **Список ORM-библиотек**

[ru.wikipedia.org](http://ru.wikipedia.org)

**C++ (4):** LiteSQL, ODB, Wt::Dbo, QxOrm

**Flex (1):** Athena Framework

**Java (24):** ActiveJDBC, Athena Framework, Carbonado, Cayenne, DataNucleus (JPOX), Ebean, EclipseLink, Enterprise Objects Framework, Fast Java Object Relation Mapping (Fjorm), Hibernate, Java Data Objects (JDO), Java Object Oriented Querying (jOOQ), Java Persistence API (JPA), Kodo, MyBatis (iBATIS), Object Relational Bridge, OpenJPA, ORMLite, QueryDSL, QuickDB ORM, TopLink, Torque, UcaOrm, RESTjee

**iOS (2):** DatabaseObjects, Core Data

**.NET (21):** ADO.NET Entity Framework, Base One Foundation Component Library, Business Logic Toolkit, Castle ActiveRecord, DatabaseObjects .NET, DataObjects.NET, Dapper, DevExpress eXpressPersistent Objects™

(XPO), ECO, EntitySpaces, iBATIS, LINQ to DB / linq2db, LLBLGen Pro, Neo, NHibernate, nHydrate, Persistor.NET, Quick Objects, Sabine.NET, Signum Framework, SubSonic

**Object Pascal (Delphi) (7):** BDE, Bold for Delphi, DB Express, ECO, EntityDAC, FireBird, InterBase

**Objective-C, Cocoa (1):** Enterprise Objects

**Perl (2):** DBIx::Class, Rose::DB

**PHP (13):** CakePHP, CodeIgniter, Doctrine, Eloquent, FuelPHP, PHPixie, Propel, Qcodo, Rocks, Redbean, Torpor, Yii, Zend Framework

**Python (7):** Django, Peewee ORM, SQLAlchemy, SQLAlchemy, SQLObject, Storm, Tryton, web2py

**Ruby (4):** ActiveRecord, Sequel, Datamapper, iBATIS

**JavaScript (1):** Sequelize

**SmallTalk (2):** DBIx::Class, Rose::DB

**Visual Basik 6.0 (1):** DatabaseObjects

## Объектные возможности:

- **Коллекции (array, set, multi-set)**
- **Типы, определяемые пользователем — UFD (User Defined Types)**
- **Структурные UFD — наследование, методы**
- **Типизированные таблицы**
- **Ссылочные типы**

## Обязательные характеристики

- Поддержка сложных объектов. В системе должна быть предусмотрена возможность создания составных объектов за счёт применения конструкторов составных объектов. Необходимо, чтобы конструкторы объектов были ортогональны, то есть любой конструктор можно было применять к любому объекту.
- Поддержка индивидуальности объектов. Все объекты должны иметь уникальный идентификатор, который не зависит от значений их атрибутов.
- Поддержка инкапсуляции. Корректная инкапсуляция достигается за счёт того, что программисты обладают правом доступа только к

спецификации интерфейса методов, а данные и реализация методов скрыты внутри объектов.

- Поддержка типов и классов. Требуется, чтобы в ООБД поддерживалась хотя бы одна концепция различия между типами и классами. (Термин «тип» более соответствует понятию абстрактного типа данных. В языках программирования переменная объявляется с указанием её типа. Компилятор может использовать эту информацию для проверки выполняемых с переменной операций на совместимость с её типом, что позволяет гарантировать корректность программного обеспечения. С другой стороны класс является неким шаблоном для создания объектов и предоставляет методы, которые могут применяться к этим объектам. Таким образом, понятие «класс» в большей степени относится ко времени исполнения, чем ко времени компиляции.)
- Поддержка наследования типов и классов от их предков. Подтип, или подкласс, должен наследовать атрибуты и методы от его супертипа, или суперкласса, соответственно.
- Перегрузка в сочетании с полным связыванием. Методы должны применяться к объектам разных типов. Реализация метода должна зависеть от типа объектов, к которым данный метод применяется. Для обеспечения этой функциональности связывание имен методов в системе не должно выполняться до времени выполнения программы.
- Вычислительная полнота. Язык манипулирования данными должен быть языком программирования общего назначения.
- Набор типов данных должен быть расширяемым. Пользователь должен иметь средства создания новых типов данных на основе набора предопределённых системных типов. Более того, между способами использования системных и пользовательских типов данных не должно быть никаких различий.

Результатом совмещения возможностей (особенностей) баз данных и возможностей объектно-ориентированных языков программирования являются Объектно-ориентированные системы управления базами данных (ООСУБД). ООСУБД позволяет работать с объектами баз данных так же, как с объектами в программировании в ОЯП. ООСУБД расширяет языки программирования, прозрачно вводя долговременные данные, управление параллелизмом, восстановление данных, ассоциированные запросы и другие возможности.

Некоторые объектно-ориентированные базы данных разработаны для плотного взаимодействия с такими объектно-ориентированными языками программирования, как Python, Java, C#, Visual Basic .NET, C++, Objective-C и Smalltalk; другие имеют свои собственные языки программирования. ООСУБД используют точно такую же модель, что и объектно-ориентированные языки программирования.

## 13.2 Абстрактные типы данных

Абстрактный тип данных (АТД) — это математическая модель для типов данных, где тип данных определяется поведением (семантикой) с точки зрения пользователя данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.

Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции. Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями. Конкретные реализации АТД называются структурами данных.

В программировании абстрактные типы данных обычно представляются в виде интерфейсов, которые скрывают соответствующие реализации типов. Программисты работают с абстрактными типами данных исключительно через их интерфейсы, поскольку реализация может в будущем измениться. Такой подход соответствует принципу инкапсуляции в объектно-ориентированном программировании. Сильной стороной этой методики является именно скрытие реализации. Раз вовне опубликован только интерфейс, то пока структура данных поддерживает этот интерфейс, все программы, работающие с заданной структурой абстрактным типом данных, будут продолжать работать. Разработчики структур данных стараются, не меняя внешнего интерфейса и семантики функций, постепенно дорабатывать реализации, улучшая алгоритмы по скорости, надёжности и используемой памяти.

Различие между абстрактными типами данных и структурами данных, которые реализуют абстрактные типы, можно пояснить на следующем примере. Абстрактный тип данных список может быть реализован при помощи массива или линейного списка с использованием различных методов динамического выделения памяти. Однако каждая реализация определяет один и тот же набор

функций, который должен работать одинаково (по результату, а не по скорости) для всех реализаций.

Абстрактные типы данных позволяют достичь модульности программных продуктов и иметь несколько альтернативных взаимозаменяемых реализаций отдельного модуля.

Примеры АТД

- Список
- Стек
- Очередь
- Ассоциативный массив
- Очередь с приоритетом

### 13.3 Объекты (основные свойства и отличительные черты)

## 14

### 14.1 Основные структуры данных

Структура данных — это контейнер, который хранит данные в определенном макете. Этот «макет» позволяет структуре данных быть эффективной в некоторых операциях и неэффективной в других.

Программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике. Для добавления, поиска, изменения и удаления данных структура данных предоставляет некоторый набор функций, составляющих её интерфейс.

Какие бывают?

**Линейные**, элементы образуют последовательность или линейный список, обход узлов линеен. Примеры: Массивы. Связанный список, стеки и очереди.

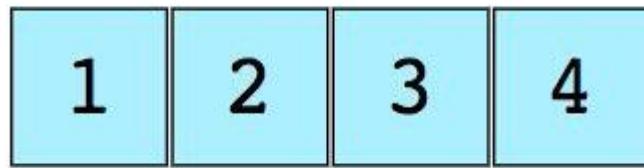
**Нелинейные**, если обход узлов нелинейный, а данные не последовательны. Пример: граф и деревья.

Основные структуры данных.

#### 14.1.1 Массивы

Массив — это самая простая и широко используемая структура данных. Другие структуры данных, такие как стеки и очереди, являются производными от массивов.

Изображение простого массива размера 4, содержащего элементы (1, 2, 3 и 4).



Каждому элементу данных присваивается положительное числовое значение (индекс), который соответствует позиции элемента в массиве. Большинство языков определяют начальный индекс массива как 0.

Бывают

- Одномерные, как показано выше.
- Многомерные, массивы внутри массивов.

Выделение памяти

- Статические
- Динамические

Основные операции

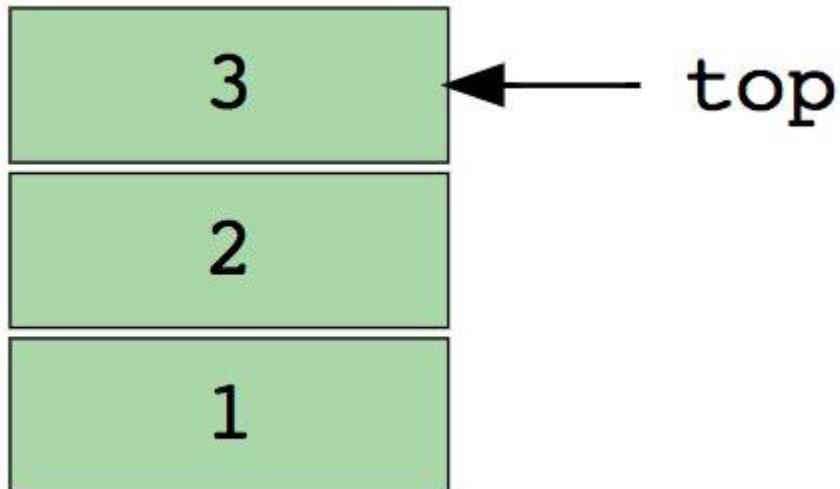
- Insert-вставляет элемент по заданному индексу
- Get-возвращает элемент по заданному индексу
- Delete-удаление элемента по заданному индексу
- Size-получить общее количество элементов в массиве

### *14.1.2 Стеки*

Стек — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Примером стека может быть куча книг, расположенных в вертикальном порядке. Для того, чтобы получить книгу, которая где-то посередине, вам нужно будет удалить все книги, размещенные на ней. Так работает метод LIFO (Last In First Out). Функция «Отменить» в приложениях работает по LIFO.

Изображение стека, в три элемента (1, 2 и 3), где 3 находится наверху и будет удален первым.



### Основные операции

- Push-вставляет элемент сверху
- Pop-возвращает верхний элемент после удаления из стека
- Size-получить общее количество элементов
- Топ-возвращает верхний элемент без удаления из стека

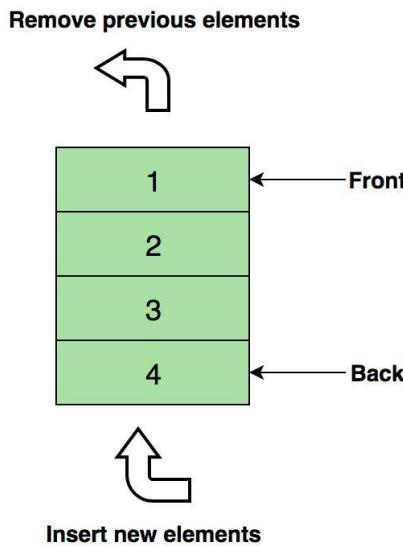
#### *14.1.3 Очереди*

Подобно стекам, очередь — хранит элемент последовательным образом.

Существенное отличие от стека – использование FIFO (First in First Out) вместо LIFO.

Пример очереди – очередь людей. Последний занял последним и будешь, а первый первым ее и покинет.

Изображение очереди, в четыре элемента (1, 2, 3 и 4), где 1 находится наверху и будет удален первым



### Основные операции

- Enqueue — вставляет элемент в конец очереди
- Dequeue () — удаляет элемент из начала очереди
- Size-получить общее количество элементов
- Top-возвращает верхний элемент без удаления

#### 14.1.4 Связанный список

Связанный список — массив где каждый элемент является отдельным объектом и состоит из двух элементов — данных и ссылки на следующий узел.

Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

- Однонаправленный, каждый узел хранит адрес или ссылку на следующий узел в списке и последний узел имеет следующий адрес или ссылку как *NULL*.  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow NULL$
- Двунаправленный, две ссылки, связанные с каждым узлом, одним из опорных пунктов на следующий узел и один к предыдущему узлу.  $NULL <- 1 <-> 2 <-> 3 > NULL$
- Круговой, все узлы соединяются, образуя круг. В конце нет *NULL*. Циклический связанный список может быть одно-или двукратным циклическим связанным списком.  $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$

### Основные операции

- InsertAtEnd — Вставка заданного элемента в конец списка
- InsertAtHead — Вставка элемента в начало списка

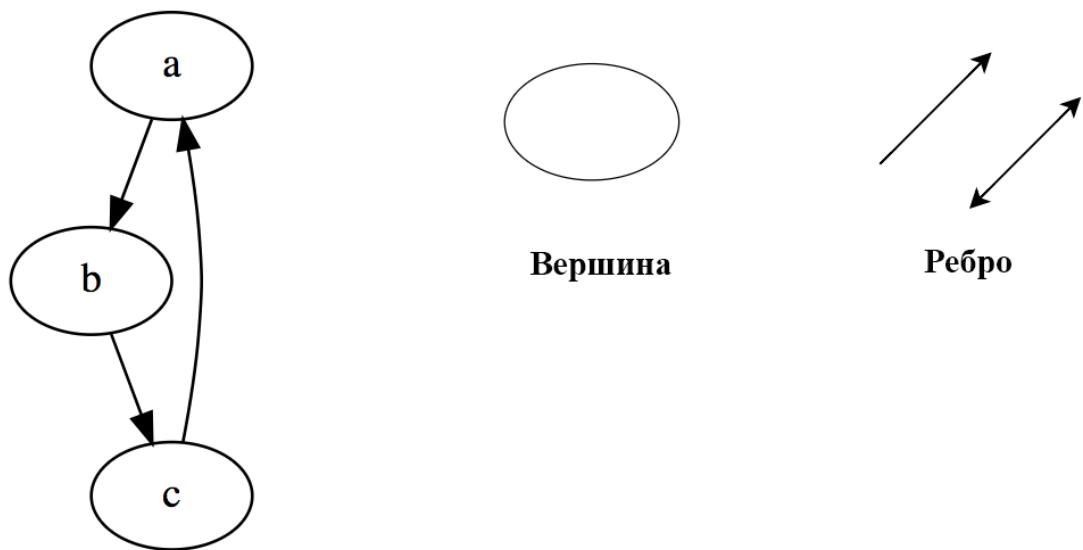
- Delete — удаляет заданный элемент из списка
- DeleteAtHead — удаляет первый элемент списка
- Search — возвращает заданный элемент из списка
- isEmpty — возвращает True, если связанный список пуст

#### *14.1.5 Ассоциативный массив*

Это абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

#### *14.1.6 Графы*

Граф-это набор узлов (вершин), которые соединены друг с другом в виде сети ребрами (дугами).



Бывают

- Ориентированный, ребра являются направленными, т.е. существует только одно доступное направление между двумя связанными вершинами.
- Неориентированные, к каждому из ребер можно осуществлять переход в обоих направлениях.
- Смешанные

Встречаются в таких формах как

- Матрица смежности. Таблица, где как столбцы, так и строки соответствуют вершинам графа. В каждой ячейке этой матрицы записывается число, определяющее наличие связи от вершины-строки к вершине-столбцу (либо наоборот). Недостатком являются требования к памяти, прямо пропорциональные квадрату количества вершин.
- Список смежности. Список, где каждой вершине графа соответствует строка, в которой хранится список смежных вершин. Такая структура данных не является таблицей в обычном понимании, а представляет собой «список списков». Это наиболее удобный способ для представления разреженных графов.
- Список рёбер. Список, где каждому ребру графа соответствует строка, в которой хранятся две вершины, инцидентные ребру. Это наиболее компактный способ представления графов, поэтому часто применяется для внешнего хранения или обмена данными.

**Граф**

```

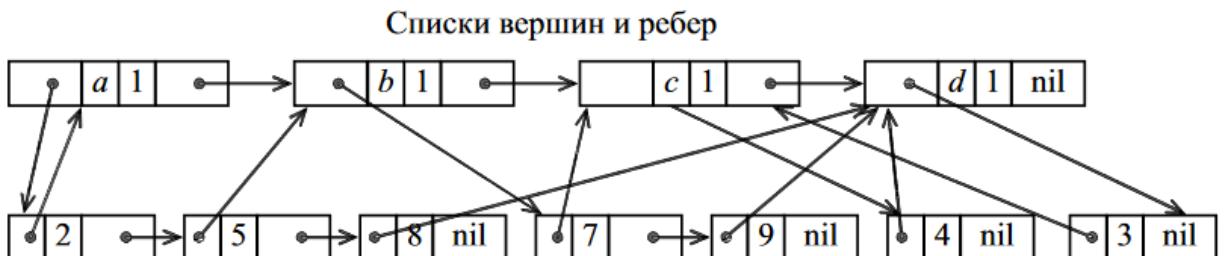
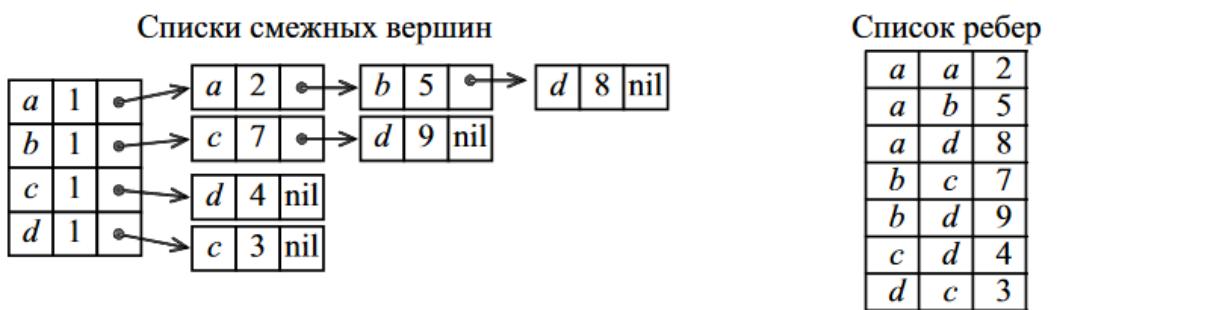
graph LR
    a((a)) -- 8 --> d((d))
    a -- 9 --> c((c))
    a -- 7 --> b((b))
    b -- 5 --> d
    c -- 4 --> d
    
```

**Матрица смежности**

	a	b	c	d
a	2	5	0	8
b	0	0	7	9
c	0	0	0	4
d	0	0	3	0

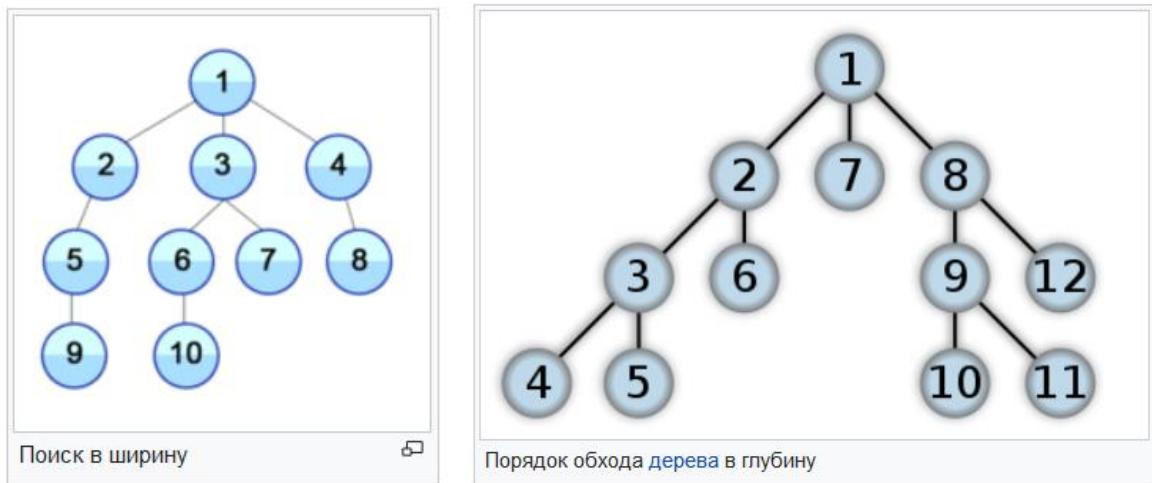
**Матрица инцидентности**

	a	b	c	d
(a, a)	2	0	0	0
(a, b)	0	5	0	0
(a, d)	0	0	0	8
(b, c)	0	0	7	0
(b, d)	0	0	0	9
(c, d)	0	0	0	4
(d, a)	0	0	3	0



Общие алгоритмы обхода графа

- Поиск в ширину – обход по уровням
- Поиск в глубину – обход по вершинам



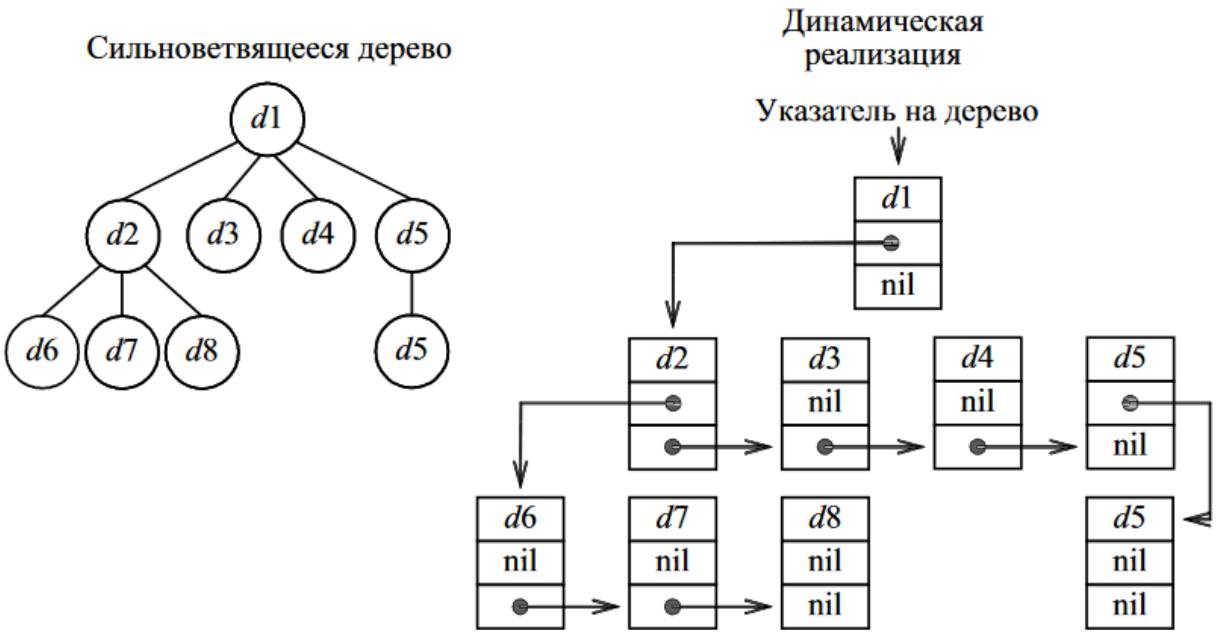
#### 14.1.7 Деревья

Дерево – это иерархическая структура данных, состоящая из узлов (вершин) и ребер (дуг). Деревья по сути связанные графы без циклов.



- Степень узла — количество исходящих дуг (или, иначе, количество поддеревьев узла).
- Концевой узел (лист) — узел со степенью 1 (то есть узел, в который ведёт только одно ребро; в случае ориентированного дерева — узел, в который ведёт только одна дуга и не исходит ни одной дуги).
- Узел ветвления — неконцевой узел.
- Уровень узла — длина пути от корня до узла. Можно определить рекурсивно: уровень корня дерева Т равен 0; уровень любого другого

узла на единицу больше, чем уровень корня ближайшего поддерева дерева  $T$ , содержащего данный узел.



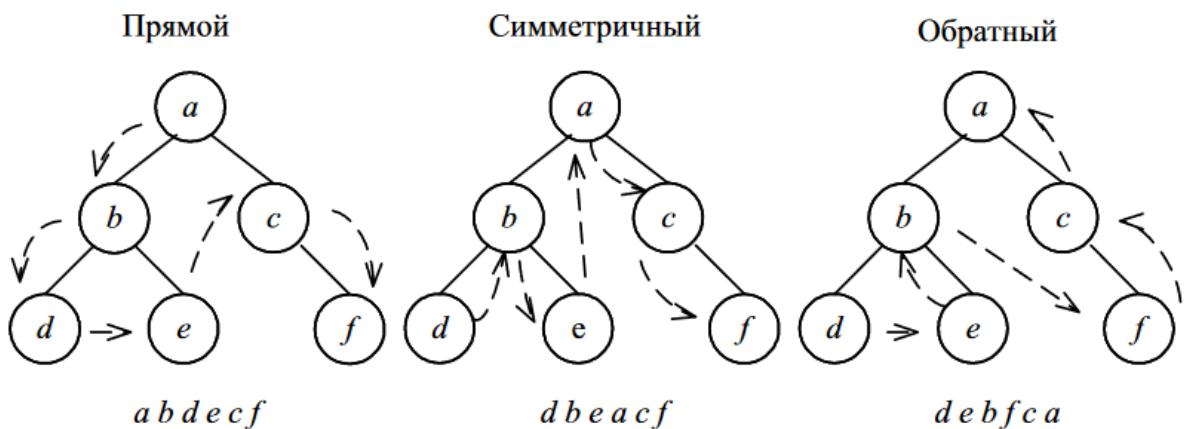
### Обход дерева

Существует несколько способов обхода (просмотра) всех вершин дерева. Три наиболее часто используемых способа обхода называются (рис. 15):

- в прямом порядке;
- в обратном порядке;
- в симметричном (внутреннем) порядке.

Все три способа обхода рекурсивно можно определить следующим образом:

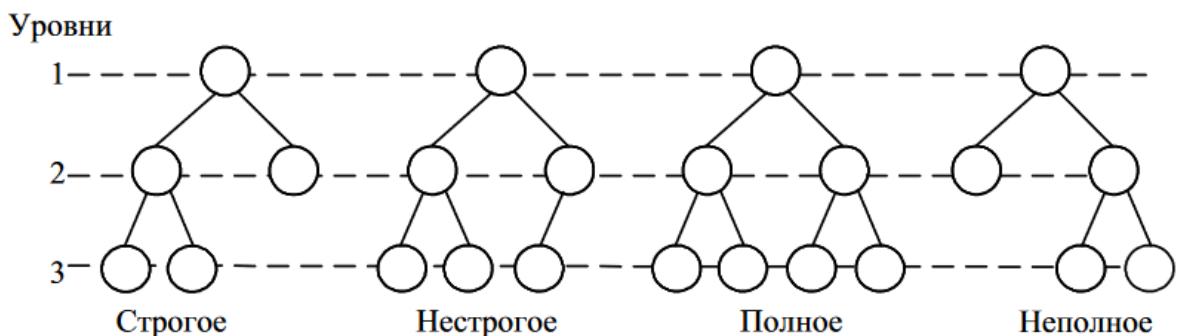
- 1) если дерево  $Tree$  является пустым деревом, то в список обхода заносится пустая запись;
- 2) если дерево  $Tree$  состоит из одной вершины, то в список обхода записывается эта вершина;
- 3) если  $Tree$  – дерево с корнем  $n$  и поддеревьями  $Tree_1, Tree_2, \dots, Tree_k$ , то:
  - при прохождении в прямом порядке сначала посещается корень  $n$ , затем в прямом порядке вершины поддерева  $Tree_1$ , далее в прямом порядке вершины поддерева  $Tree_2$  и т. д. Последними в прямом порядке посещаются вершины поддерева  $Tree_k$ ;
  - при прохождении в обратном порядке сначала посещаются в обратном порядке вершины поддерева  $Tree_1$ , далее последовательно в обратном порядке посещаются вершины поддеревьев  $Tree_2, \dots, Tree_k$ . Последним посещается корень  $n$ ;



### *14.1.7.1 Бинарное дерево*

Двоичные (бинарные) деревья – это деревья со степенью не более двух.

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка.



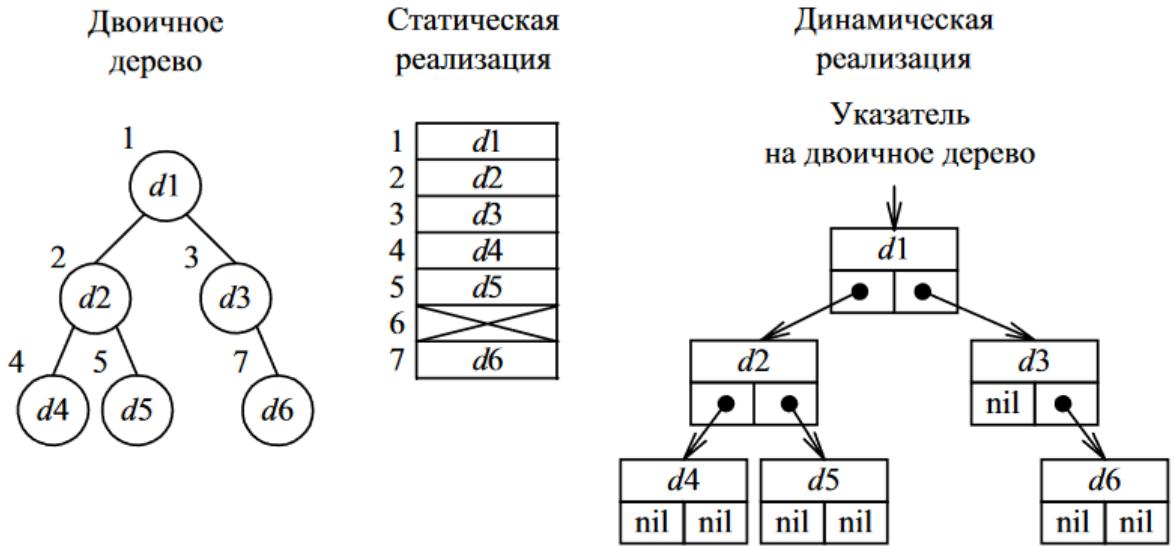
**Рис. 16. Двоичное дерево**

- с т р о г и е – вершины дерева имеют степень нуль (у листьев) или два (у узлов);
  - н е с т р о г и е – вершины дерева имеют степень нуль (у листьев), один или два (у узлов).

В общем случае на  $k$ -м уровне двоичного дерева может быть до  $2^{k-1}$  вершин.

Двоичное дерево, содержащее только полностью заполненные уровни (т. е.  $2^{k-1}$  вершин на каждом  $k$ -м уровне), называется полным.

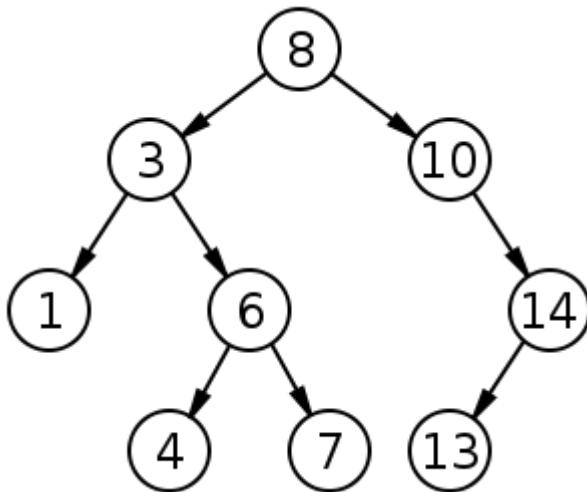
Двоичное дерево можно реализовывать как статическую структуру данных в виде одномерного массива, а можно как динамическую структуру – в виде списка (рис. 17).



Вейвлет преобразование.

#### 14.1.7.2 Дерево Бинарного Поиска

Бинарное дерево поиска — дерево в котором узлы располагаются таким образом, что каждый узел с меньшим значением (относительно родителя) находится в левой части дерева, соответственно с большим — в правой.



Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого неглубокого листа (в красно-черных деревьях). В сбалансированном

бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список, и операции поиска, вставки и удаления будут выполнятся за линейное время. Поэтому балансировка дерева крайне важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

#### *14.1.7.3 Сбалансированное дерево*

Дерево поиска с минимальной высотой как раз и называется сбалансированным, т.е. таким, в котором высота левого и правого поддеревьев отличаются не более чем на единицу.

Сбалансированное оно потому, что в таком дереве в среднем требуется наименьшее количество операций для поиска.

Чтобы дерево имело минимальную высоту, количество узлов левого и правого поддеревьев должны максимально приближаться друг к другу. Построим дерево по этому принципу: середина каждого подраздела массива становится корневым узлом, а левая и правая части — соответствующими для него поддеревьями. Т.к. массив отсортирован, то полученное дерево соответствует определению бинарного дерева поиска.

Алгоритм получился примерно такой:

- выбрать средний элемент массива в качестве корня
- начинать создавать левое поддерево из левой части массива
- начинать создавать правое поддерево из правой части массива
- повторить рекурсивно, пока не закончатся элементы в массиве

```
1  /**
2   * Конструктор узла дерева.
3   * Каждый узел имеет 2 ссылки: левую и правую части поддерева.
4   */
5  class Node {
6      constructor(data) {
7          this.data = data;
8          this.left = null;
9          this.right = null;
10     }
11 }
12
13 /**
14  * По переданному отсортированному массиву
15  * создается сбалансированное дерево бинарного поиска
16  */
17 function createBalancedTree(arr) {
18     return _createBalancedTree(arr, 0, arr.length - 1);
19 }
20
21 /**
22  * Рекурсивно создает узлы для каждой «половинки» массива:
23  * левые и правые части превращаются в соответствующие поддеревья
24  */
25 function _createBalancedTree(arr, start, end) {
26     if (end < start) {
27         return null;
28     }
29     const mid = Math.floor((start + end) / 2);
30     const node = new Node(arr[mid]);
31
32     node.left = _createBalancedTree(arr, start, mid - 1);
33     node.right = _createBalancedTree(arr, mid + 1, end);
34
35     return node;
36 }
```

Единственное на что нужно обратить внимание — смещение на единицу при выборе границ частей массива.

<https://medium.com/@vitkarpov/cracking-the-coding-interview-4-2-9567d6986853>

#### 14.1.7.4 N дерево

N-арные деревья определяются по аналогии с двоичным деревом. Для них также есть ориентированные и неориентированные случаи, а также соответствующие абстрактные структуры данных.

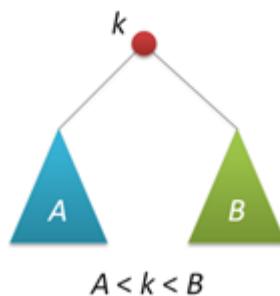
- N-арное дерево (неориентированное) — это дерево (обычное, неориентированное), в котором степени вершин не превосходят  $N+1$ .
- N-арное дерево (ориентированное) — это ориентированное дерево, в котором исходящие степени вершин (число исходящих рёбер) не превосходят  $N$ .

#### 14.1.7.5 AVL дерево (~~сбалансированное дерево~~)

Сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

AVL — аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса.

AVL-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в AVL-дереве можно использовать стандартный алгоритм. Для простоты дальнейшего изложения будем считать, что все ключи в дереве целочисленны и не повторяются.



Особенностью AVL-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота  $h$  AVL-дерева с  $n$  ключами лежит в диапазоне от  $\log_2(n + 1)$  до  $1.44 \log_2(n + 2) - 0.328$ . А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов)

линейно зависят от его высоты, то получаем гарантированную логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве. Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших  $n$  хотя и является пренебрежимо малой, но остается не равной нулю.

<https://habr.com/ru/post/150732/>



#### 14.1.7.6 В-дерево

В-дерево (по-русски произносится как Би-дерево) — структура данных, дерево поиска. С точки зрения внешнего логического представления, сбалансированное, сильно ветвистое дерево. Часто используется для хранения данных во внешней памяти.

<https://youtu.be/WXxetwePSRk> (Volodya Mozhenkov - В-дерево)

Хорошо для хранения на жестком диске, тк уровень (размер 1 ячейки) можно подогнать под размер страницы на жестком диске, что позволит избежать лишнего перехода от страницы к странице.

В дерево хорошо, когда данные нужно добавлять редко, тк каждое добавление в худшем случае может стоить перестроением. Но если добавление редкое – то ок.

Основной недостаток В-деревьев состоит в отсутствии для них эффективных средств выборки данных (т.е. метода обхода дерева), упорядоченных по отличному от выбранного ключа.

Сбалансированность означает, что длина любых двух путей от корня до листьев различается не более, чем на единицу.

Ветвистость дерева — это свойство каждого узла дерева ссылаться на большое число узлов-потомков.

С точки зрения физической организации В-дерево представляется как мультисписочная структура страниц памяти, то есть каждому узлу дерева соответствует блок памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

Структура В-дерева применяется для организации индексов во многих современных СУБД и структурирования информации на жёстком диске.

Как мы уже видели, очень эффективным является хранение множества данных в виде дерева. Поэтому в качестве типового способа организации внешней памяти стало В-дерево, которое обеспечивает при своем обслуживании относительно небольшое количество обращений к внешней памяти (рис. 19).

В-дерево представляет собой дерево поиска степени  $m$ , характеризующееся следующими свойствами:

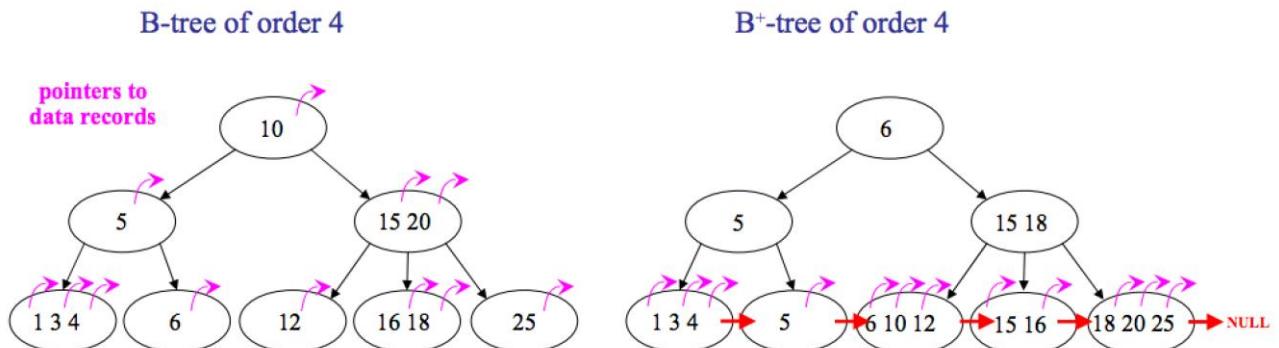
- 1) корень либо является листом, либо имеет не менее двух потомков;
- 2) каждый узел, кроме корня и листьев, имеет от  $(m/2)$  до  $m$  потомков;
- 3) все пути от корня до любого листа имеют одинаковую длину.

#### Основные достоинства

- Во всех случаях полезное использование пространства вторичной памяти составляет свыше 50 %. С ростом степени полезного использования памяти не происходит снижения качества обслуживания.
- Произвольный доступ к записи реализуется посредством малого количества подопераций (обращения к физическим блокам).
- В среднем достаточно эффективно реализуются операции включения и удаления записей; при этом сохраняется естественный порядок ключей с целью последовательной обработки, а также соответствующий баланс дерева для обеспечения быстрой произвольной выборки.
- Неизменная упорядоченность по ключу обеспечивает возможность эффективной пакетной обработки (запуск «заданий», которые могут выполняться без взаимодействия с конечным пользователем или могут запускаться по расписанию, если позволяют ресурсы).

#### 14.1.7.7 B+ дерево

- A B<sup>+</sup>-tree can be viewed as a B-tree in which each node contains only keys (not pairs), and to which an additional level is added at the bottom with linked leaves



Преимущества деревьев B +:

Поскольку деревья B + не имеют данных, связанных с внутренними узлами, на одной странице памяти может поместиться больше ключей. Следовательно, для доступа к данным, которые находятся на листовом узле, потребуется меньше промахов в кэше.

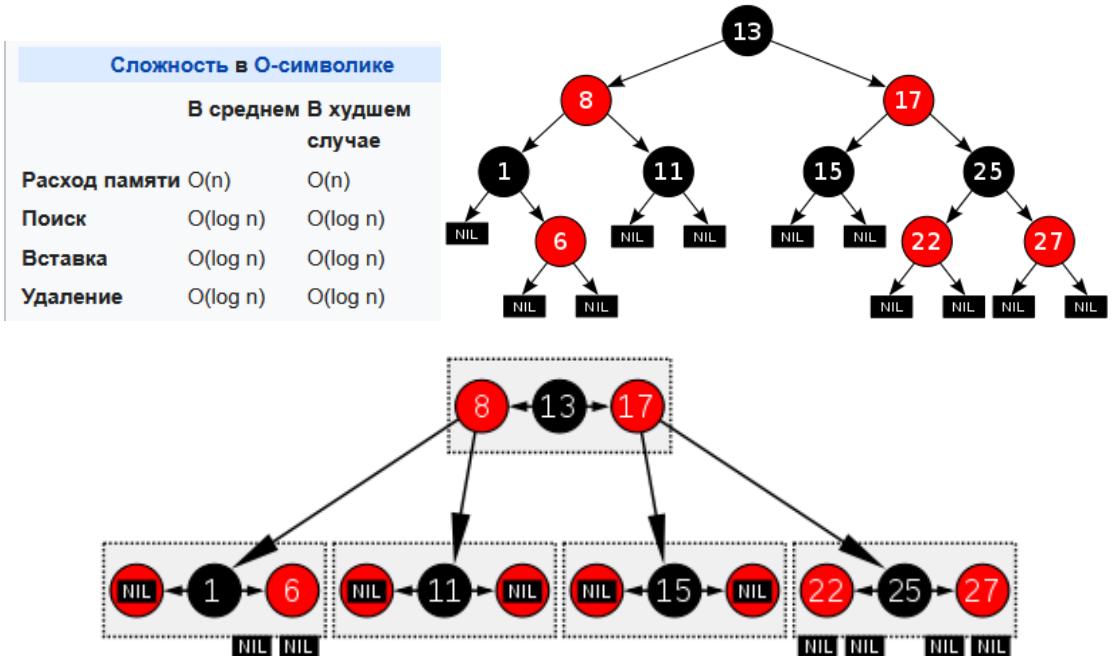
Листовые узлы деревьев B + связаны, поэтому для полного сканирования всех объектов в дереве требуется всего один линейный проход через все листовые узлы. B-дерево, с другой стороны, потребует обхода каждого уровня в дереве. Этот обход всего дерева, вероятно, будет включать больше пропусков кеша, чем линейный обход листьев B +.

Преимущество B-деревьев: Поскольку B-деревья содержат данные с каждым ключом, часто используемые узлы могут находиться ближе к корню и, следовательно, к ним можно получить доступ быстрее.

Но B+ деревья хороши, когда нужно отобразить в оперативной памяти большой объем жесткого диска.

#### 14.1.7.8 Красно-чёрное дерево

Один из видов самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».



Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут цвета. При этом:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья — чёрные и не содержат данных.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Благодаря этим ограничениям, путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Чтобы понять, как это работает, достаточно рассмотреть эффект свойств 4 и 5 вместе. Пусть для красно-чёрного дерева Т число чёрных узлов от корня до листа равно В. Тогда кратчайший возможный путь до любого листа содержит В узлов и все они чёрные. Более длинный возможный путь может быть построен путём включения красных узлов. Однако, благодаря п.4 в дереве не может быть двух красных узлов подряд, а согласно пп. 2 и 3, путь начинается и кончается

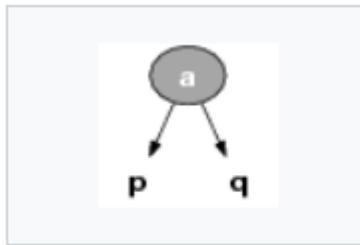
чёрным узлом. Поэтому самый длинный возможный путь состоит из 2В-1 узлов, попеременно красных и чёрных.

<https://youtu.be/n7Y2karbxF4> (Volodya Mozhenkov - Красно-Чёрные Деревья)

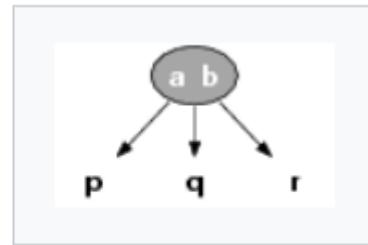
#### 14.1.7.9 R-дерево

#### 14.1.7.10 2-3 деревья

Структура данных, являющаяся В-деревом степени 1, каждый узел (страница) которого имеет либо два потомка и одно поле, либо три потомка и два поля. Листовые вершины являются исключением — у них нет детей (но может быть одно или два поля). 2-3 деревья сбалансированы, то есть каждое левое, правое и центральное поддерево имеет одну и ту же высоту, и, таким образом, содержат равные (или почти равные) объёмы данных



2-вершина



3-вершина

#### Свойства

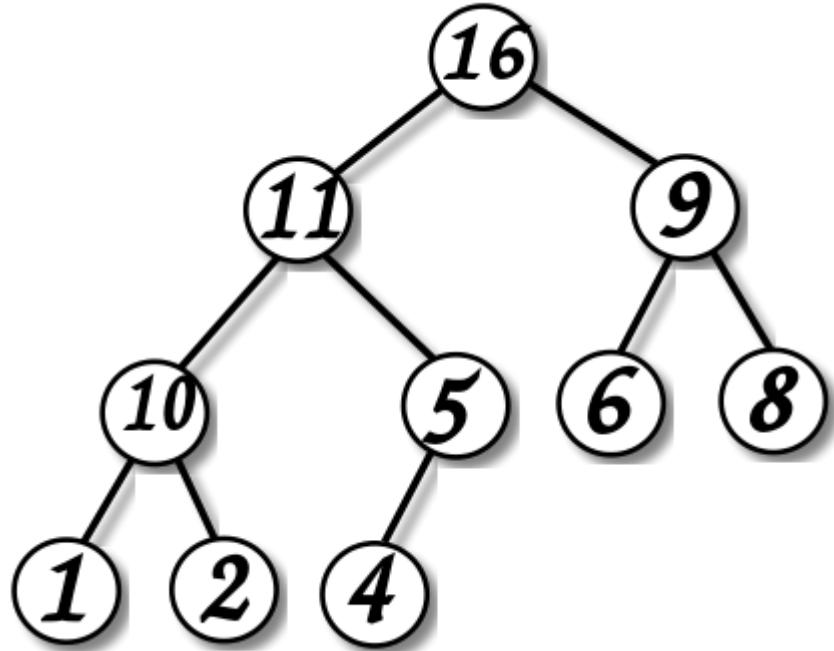
- Все нелистовые вершины содержат одно поле и 2 поддерева или 2 поля и 3 поддерева.
- Все листовые вершины находятся на одном уровне (на нижнем уровне) и содержат 1 или 2 поля.
- Все данные отсортированы (по принципу [двоичного дерева поиска](#)).
- Поле в 2-вершине, как и в двоичном дереве поиска, делит пространство возможных значений на два диапазона:  $(-\infty, a)$  и  $[a, +\infty)$
- Поля в 3-вершине делят пространство возможных значений на три диапазона:  $(-\infty, a)$ ,  $[a, b)$  и  $[b, +\infty)$

#### 14.1.7.11 Двоичная куча

Такое двоичное дерево, для которого выполнены три условия:

- Значение в любой вершине не меньше, чем значения её потомков.
- Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой.
- Последний слой заполняется слева направо без «дырок».

Существуют также кучи, где значение в любой вершине, наоборот, не больше, чем значения её потомков. Такие кучи называются min-heap, а кучи, описанные выше — max-heap. В дальнейшем рассматриваются только max-heap. Все действия с min-heap осуществляются аналогично.

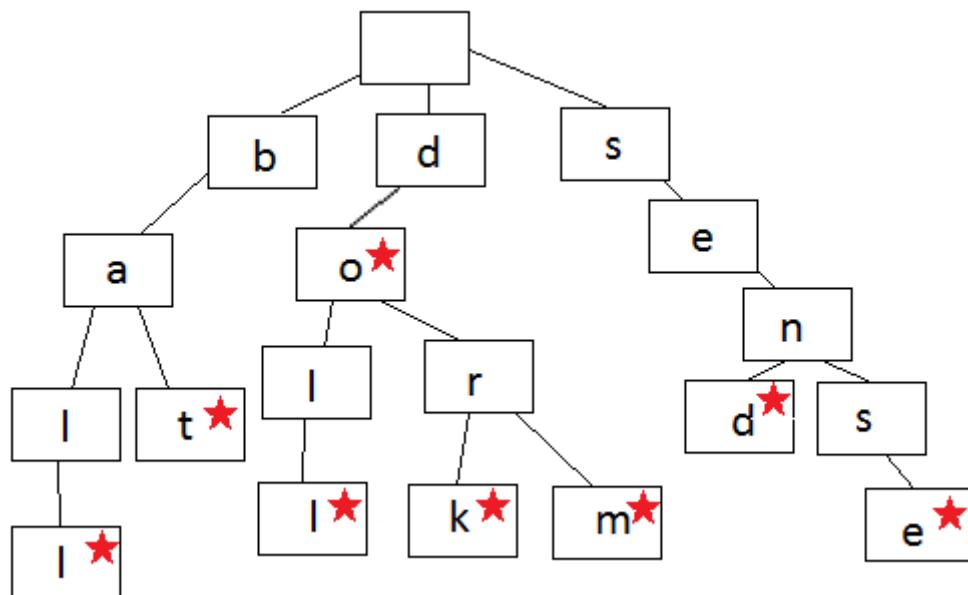


Удобная структура данных для сортирующего дерева — массив  $A$ , у которого первый элемент,  $A[1]$  — элемент в корне, а потомками элемента  $A[i]$  являются  $A[2i]$  и  $A[2i+1]$  (при нумерации элементов с первого). При нумерации элементов с нулевого, корневой элемент —  $A[0]$ , а потомки элемента  $A[i]$  —  $A[2i+1]$  и  $A[2i+2]$ . При таком способе хранения условия 2 и 3 выполнены автоматически.

<b>16</b>	<b>11</b>	<b>9</b>	<b>10</b>	<b>5</b>	<b>6</b>	<b>8</b>	<b>1</b>	<b>2</b>	<b>4</b>
-----------	-----------	----------	-----------	----------	----------	----------	----------	----------	----------

#### 14.1.8 Префиксное дерево

Префиксное (нагруженное) дерево — это разновидность дерева поиска. Оно хранит данные в метках, каждая из которых представляет собой узел на дереве. Такие структуры часто используют, чтобы хранить слова и выполнять быстрый поиск по ним — например, для функции автозаполнения.



Каждый узел в языковом префиксном дереве содержит одну букву слова. Чтобы составить слово, нужно следовать по ветвям дерева, проходя по одной букве за раз. Дерево начинает ветвиться, когда порядок букв отличается от других имеющихся в нем слов или когда слово заканчивается. Каждый узел содержит букву (данные) и булево значение, которое указывает, является ли он последним в слове.

Посмотрите на иллюстрацию и попробуйте составить слова. Всегда начинайте с корневого узла вверху и спускайтесь вниз. Это дерево содержит следующие слова: ball, bat, doll, do, dork, dorm, send, sense.

#### 14.1.9 Hash map

Что называется хешированием?

<https://www.youtube.com/watch?v=xXaqBe78AfI>

## **14.2 Алгоритмы обработки данных**

## **14.3 Алгоритмы поиска данных**

## **14.4 Сжатия данных**

Преобразование данных, производимое с целью уменьшения занимаемого ими объёма.

Сжатие основано на устраниении избыточности, содержащейся в исходных данных. Простейшим примером избыточности является повторение в тексте фрагментов. Подобная избыточность обычно устраняется заменой повторяющейся последовательности ссылкой на уже закодированный фрагмент с указанием его длины. Другой вид избыточности связан с тем, что некоторые значения в сжимаемых данных встречаются чаще других. Сокращение объёма данных достигается за счёт замены часто встречающихся данных короткими кодовыми словами, а редких — длинными. Сжатие данных, не обладающих свойством избыточности (например, случайный сигнал или белый шум, зашифрованные сообщения), принципиально невозможно без потерь.

Сжатие без потерь позволяет полностью восстановить исходное сообщение, так как не уменьшает в нем количество информации, несмотря на уменьшение длины.

Коэффициент сжатия — основная характеристика алгоритма сжатия. Она определяется как отношение объёма исходных несжатых данных к объёму сжатых данных, то есть:  $k = \frac{S_o}{S_c}$ , где  $k$  — коэффициент сжатия,  $S_o$  — объём исходных данных, а  $S_c$  — объём сжатых. Таким образом, чем выше коэффициент сжатия, тем алгоритм эффективнее.

Иногда сжатие происходит за счет ограничений устройств ввода-вывода человека. Например, глаз не различит разницу в 2 соседних пикселях, если составляющая одного из цветов отличается на 1. Так же и с музыкой, свыше 20 кГц большая часть не услышит, поэтому все, что выше этой частоты можно урезать.

Так как алгоритмы сжатия и восстановления работают в паре, имеет значение соотношение системных требований к ним. Нередко можно, усложнив один алгоритм, значительно упростить другой. Таким образом, возможны три варианта:

- Алгоритм сжатия требует больших вычислительных ресурсов, нежели алгоритм восстановления. Это наиболее распространённое

соотношение, характерное для случаев, когда однократно сжатые данные будут использоваться многократно. В качестве примера можно привести цифровые аудио- и видеопроигрыватели.

- Алгоритмы сжатия и восстановления требуют приблизительно равных вычислительных ресурсов. Наиболее приемлемый вариант для линий связи, когда сжатие и восстановление происходит однократно на двух её концах (например, в цифровой телефонии).
- Алгоритм сжатия существенно менее требователен, чем алгоритм восстановления. Такая ситуация характерна для случаев, когда процедура сжатия реализуется простым, часто портативным, устройством, для которого объём доступных ресурсов весьма критичен, например, космический аппарат или большая распределённая сеть датчиков. Это могут быть также данные, распаковка которых требуется в очень малом проценте случаев, например запись камер видеонаблюдения.

#### *14.4.1 Алгоритм Хаффмана*

Идея алгоритма состоит в следующем: зная вероятности появления символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана.

Пример ниже. Каждый раз выбирается 2 символа с наименьшими частотами и заменяется на абстрактный символ с суммой частот.

Символ	A	Б	В	Г	Д
Частота	15	7	6	6	5

1.

Символ	А	Б	В	Г	Д	$x_1$
Частота	15	7	6	9	11	

[ ]



2.

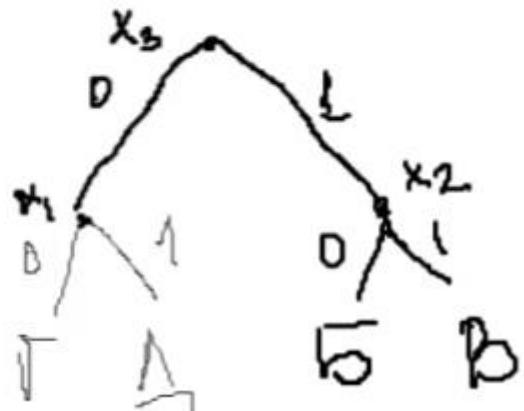
Символ	А	Б	В	Г	Д	$x_1$	$x_2$
Частота	15	11	13	9	11		13

[ ]



3.

Символ	А	Б	В	Г	Д	$x_1$	$x_2$	$x_3$
Частота	15	10	10	10	10	10	10	24



4.

Символ	А	Б	В	Г	Д	$x_1$	$x_2$	$x_3$
Частота	10	10	10	10	10	10	10	24



5.

Символ	А	Б	В	Г	Д	$x_1$	$x_2$	$x_3$
Частота	10	10	10	10	10	10	10	24

А - L  
 Б 011  
 В 010  
 Г 000  
 Д 001



6.

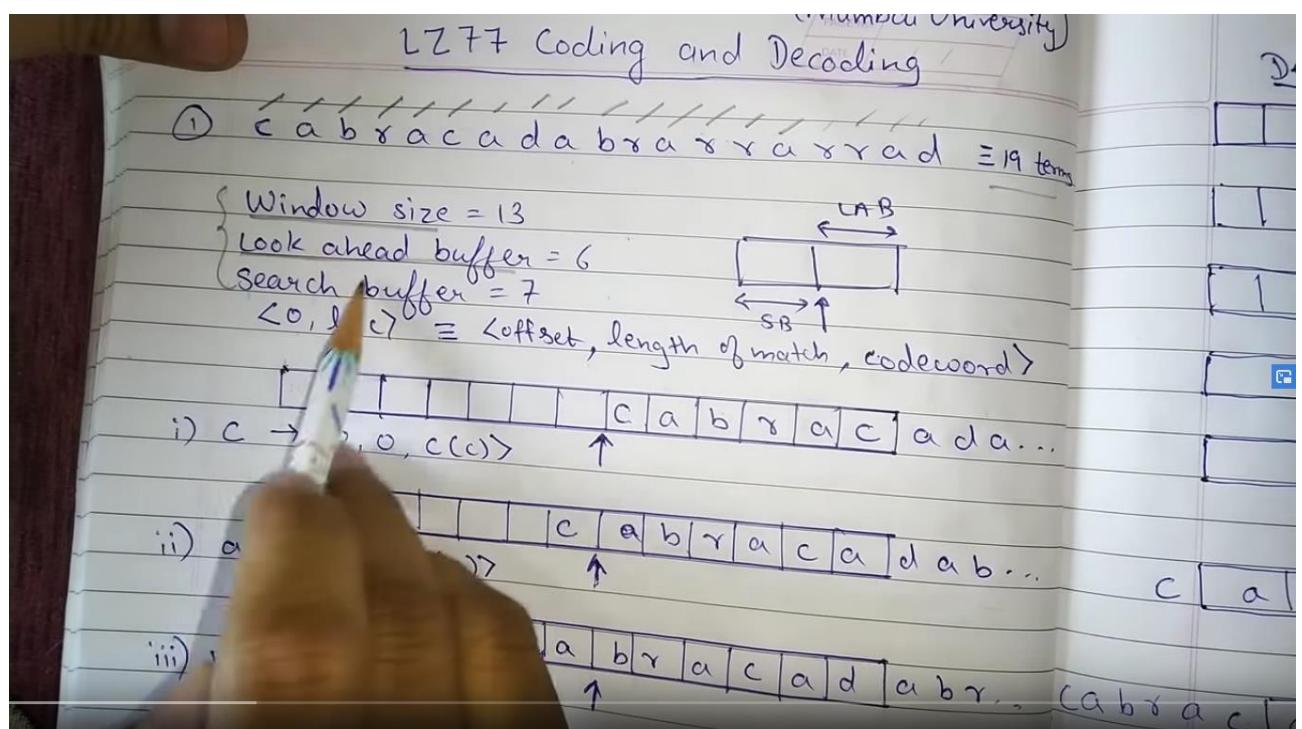
## 14.4.2 ZIP (LZ77)

Наиболее часто в ZIP используется алгоритм сжатия Deflate. Это алгоритм сжатия без потерь, использующий комбинацию алгоритмов LZ77 и Хаффмана.

Как работает LZ77 – см далее

<https://www.youtube.com/watch?v=cSyK2iCqr4w>

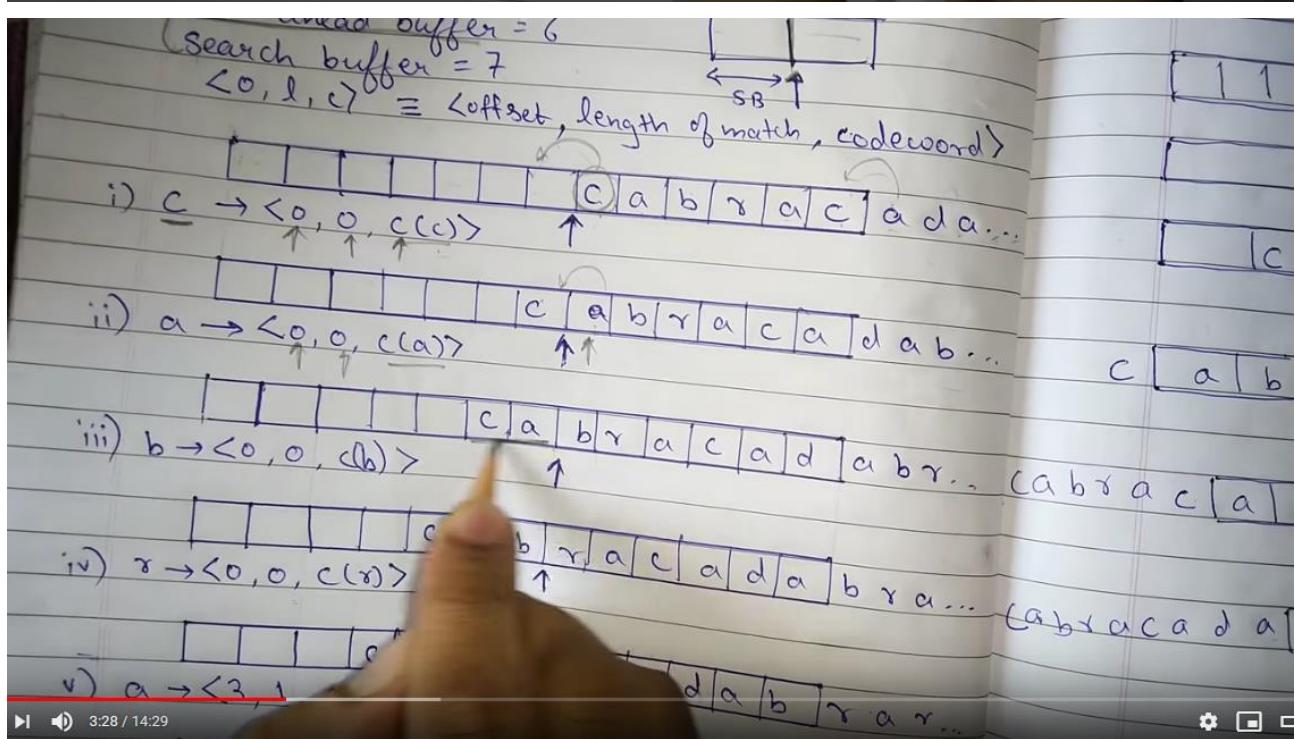
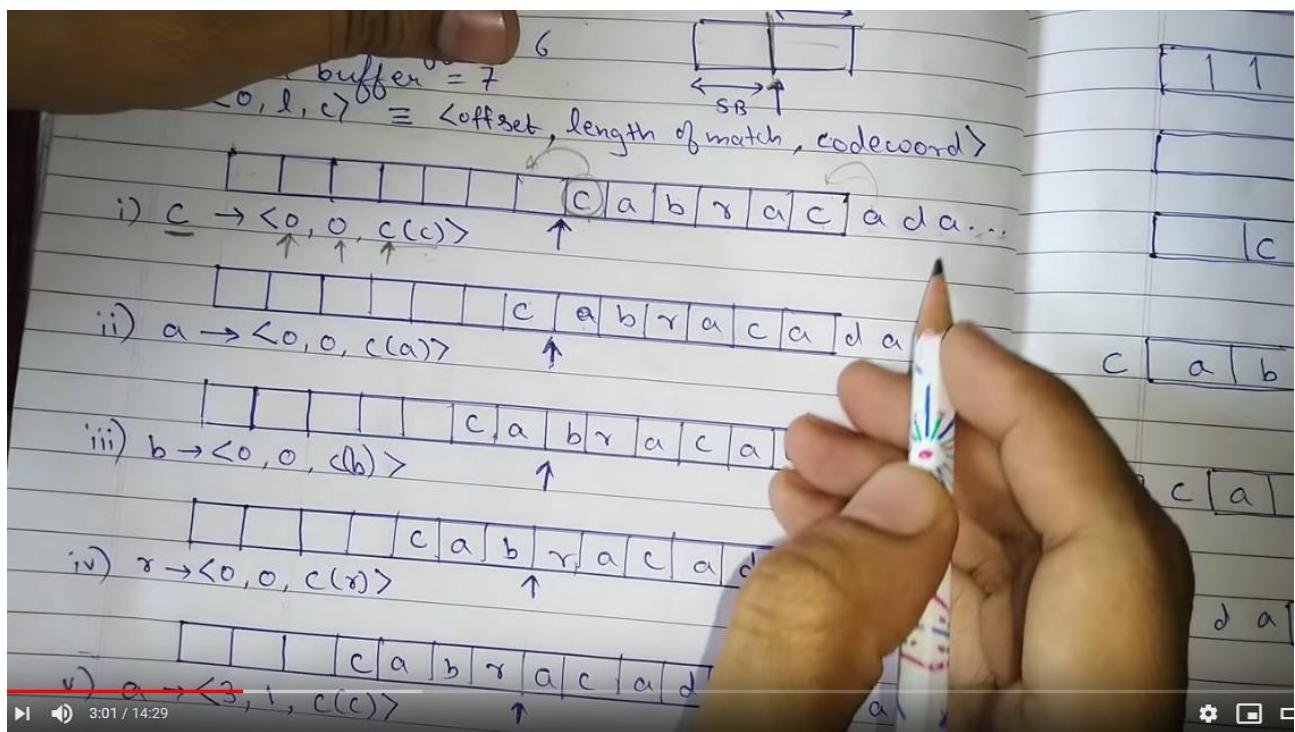
Есть буффер, он состоит из 2х частей: look ahead (LA) & search (S) слева и справа соответственно.

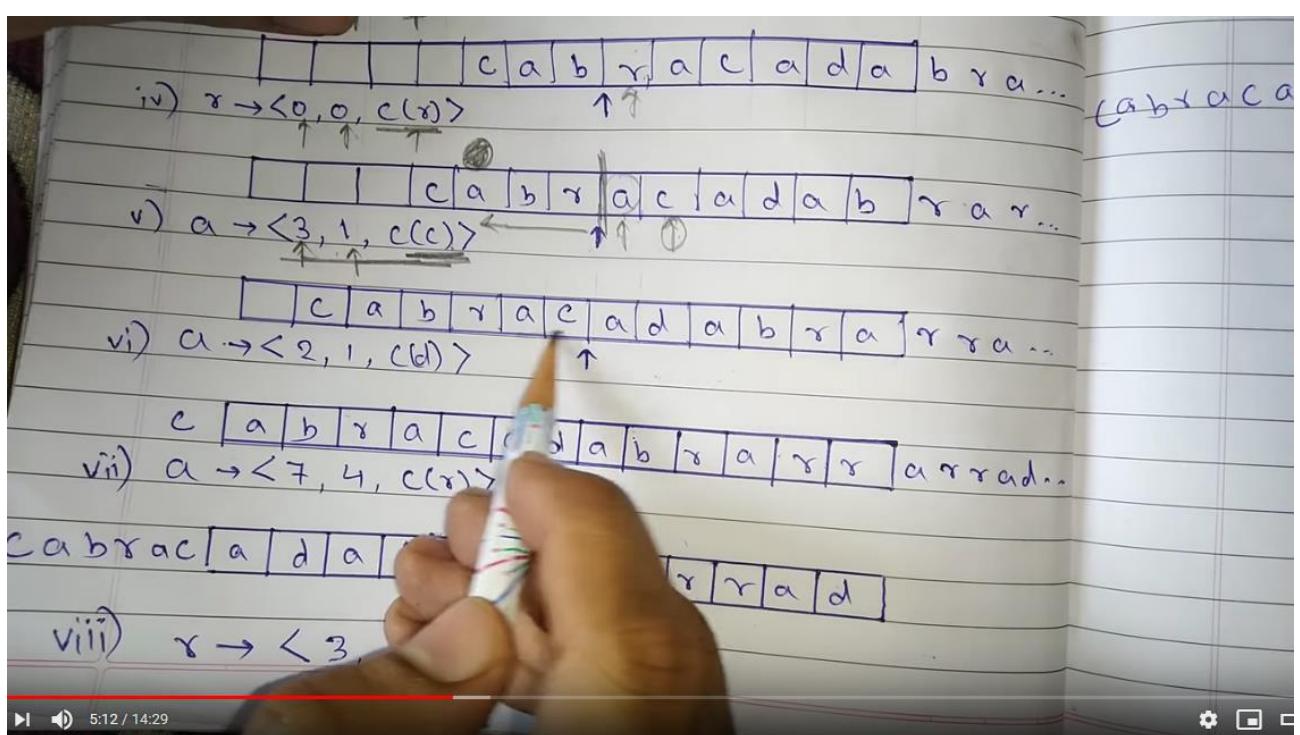
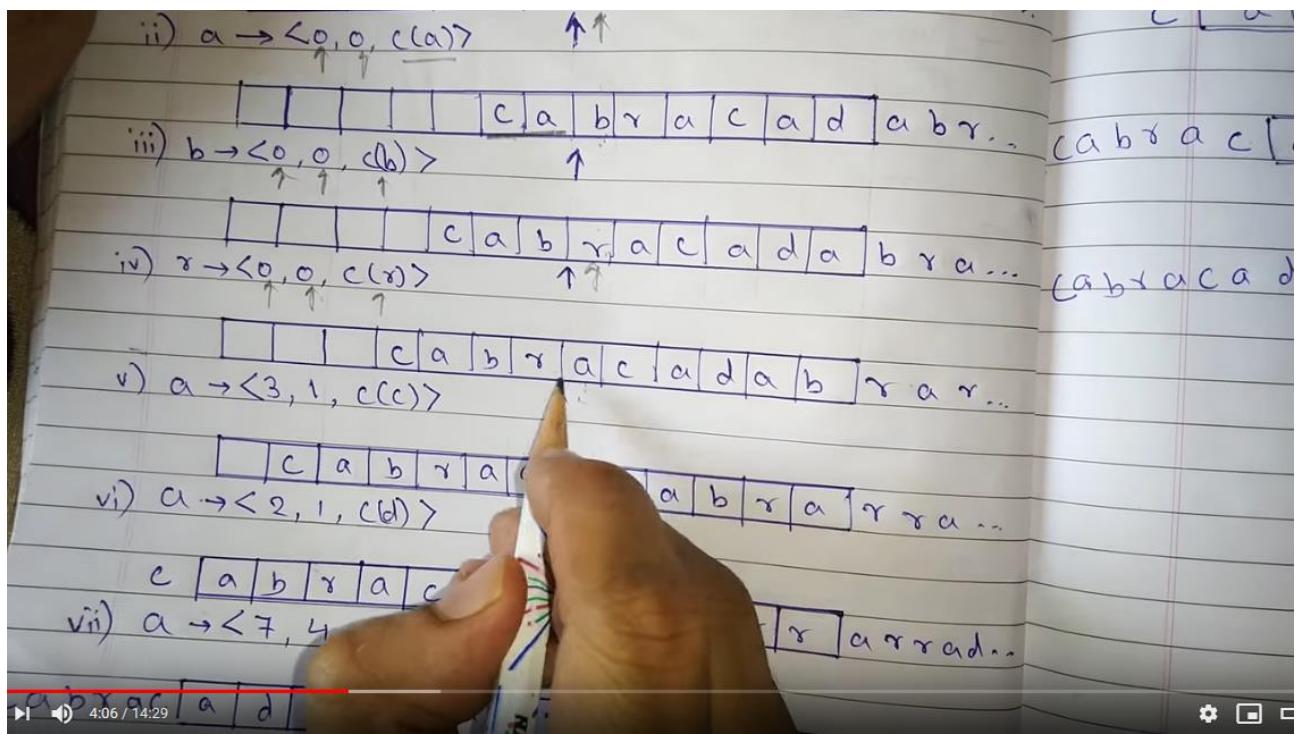


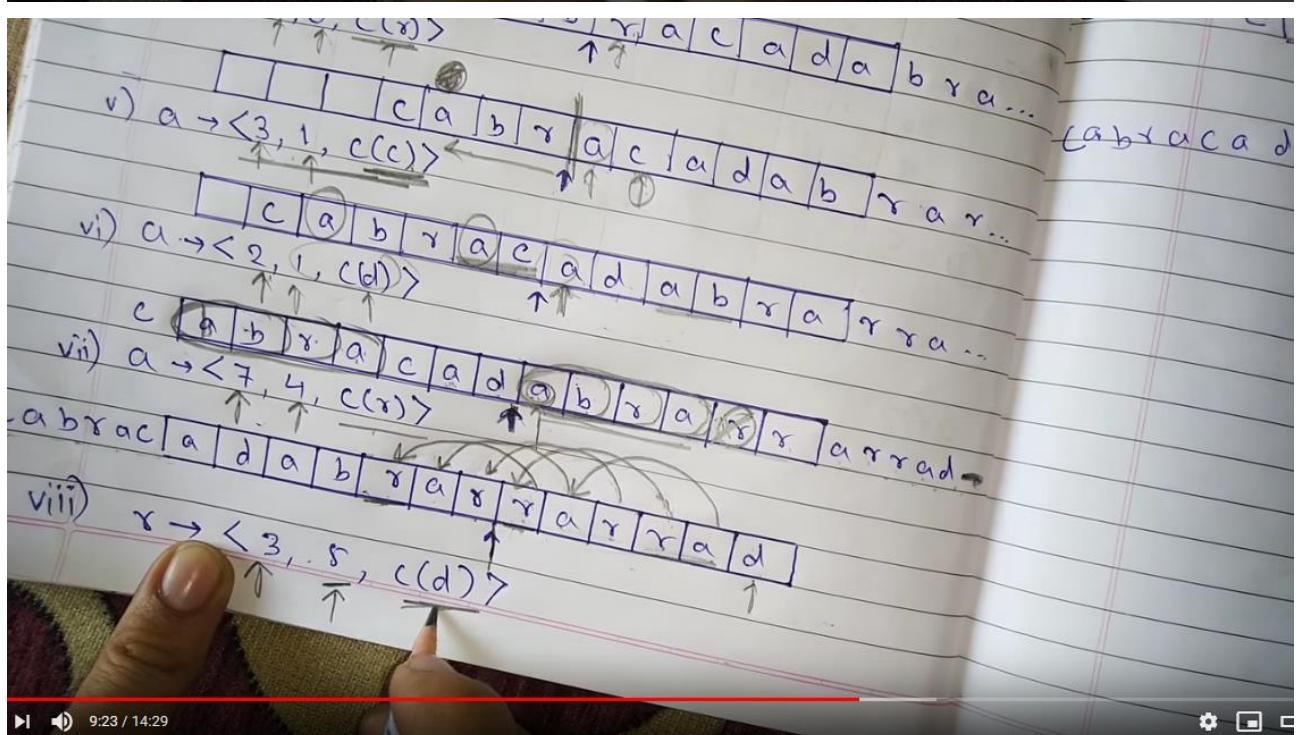
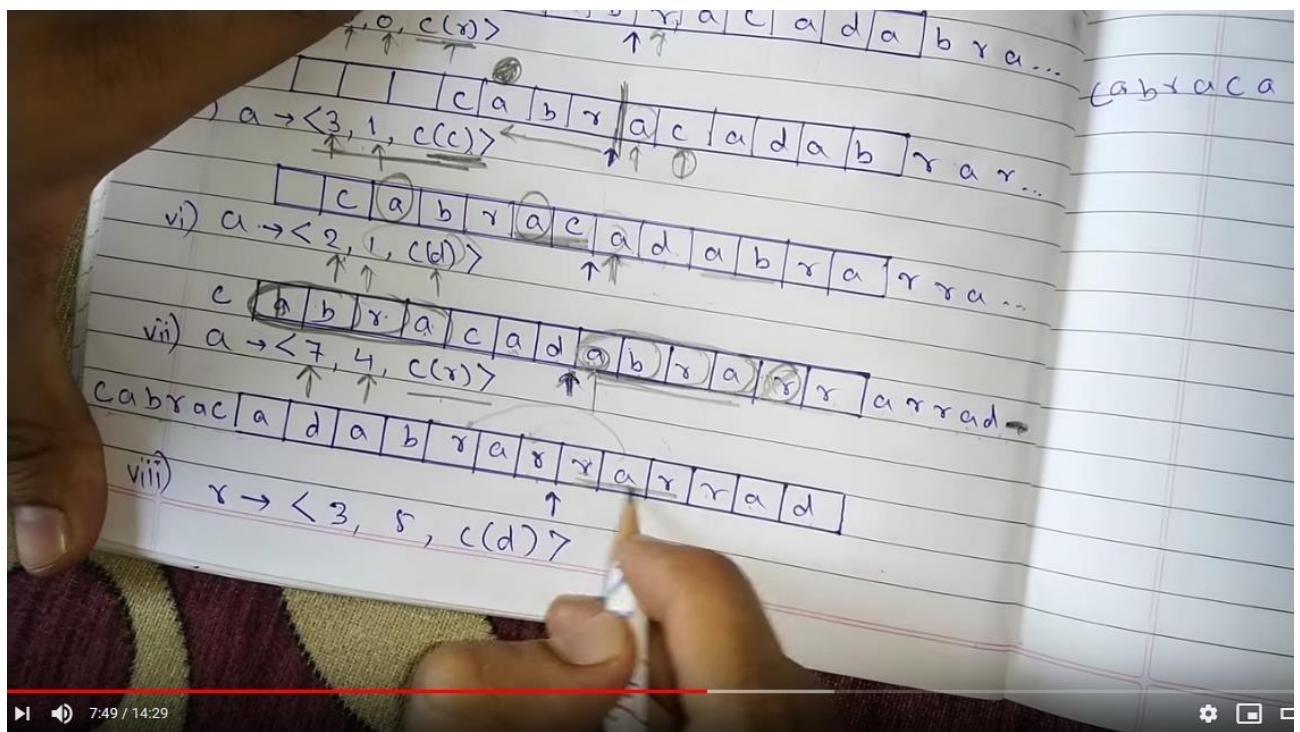
Стрелка – граница между LA & S.

Команды имеют вид  $\langle$ offset, length, letter $\rangle$ . Хз зачем в букве индус пишет с(letter). Далее смотри по пикчам.

#### *14.4.2.1 Кодирование*





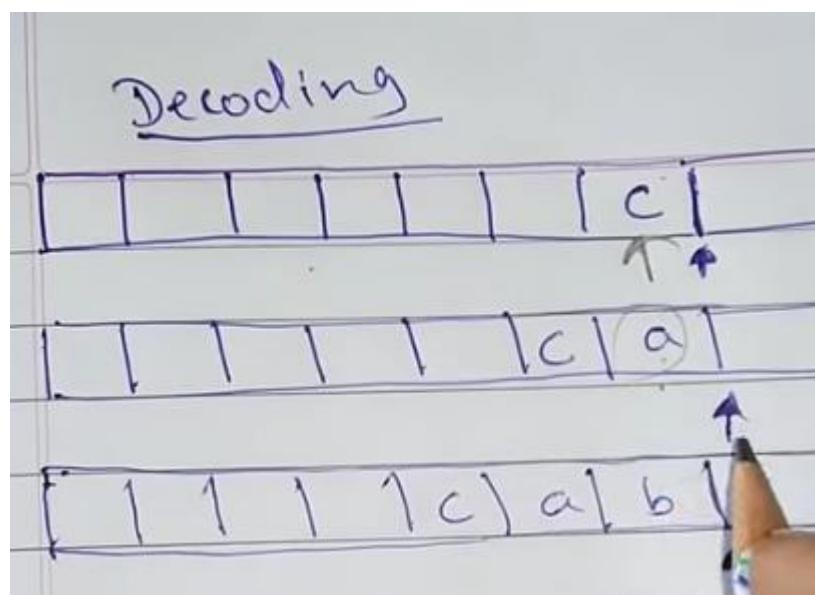


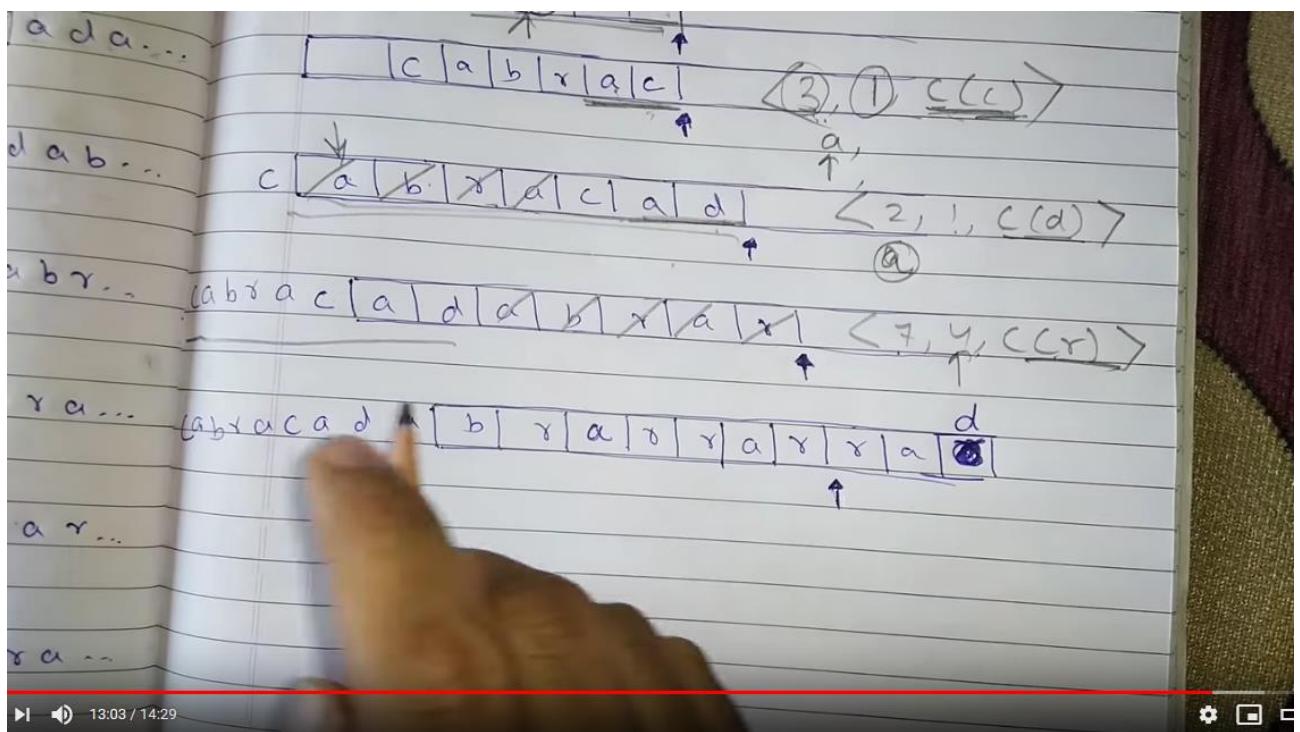
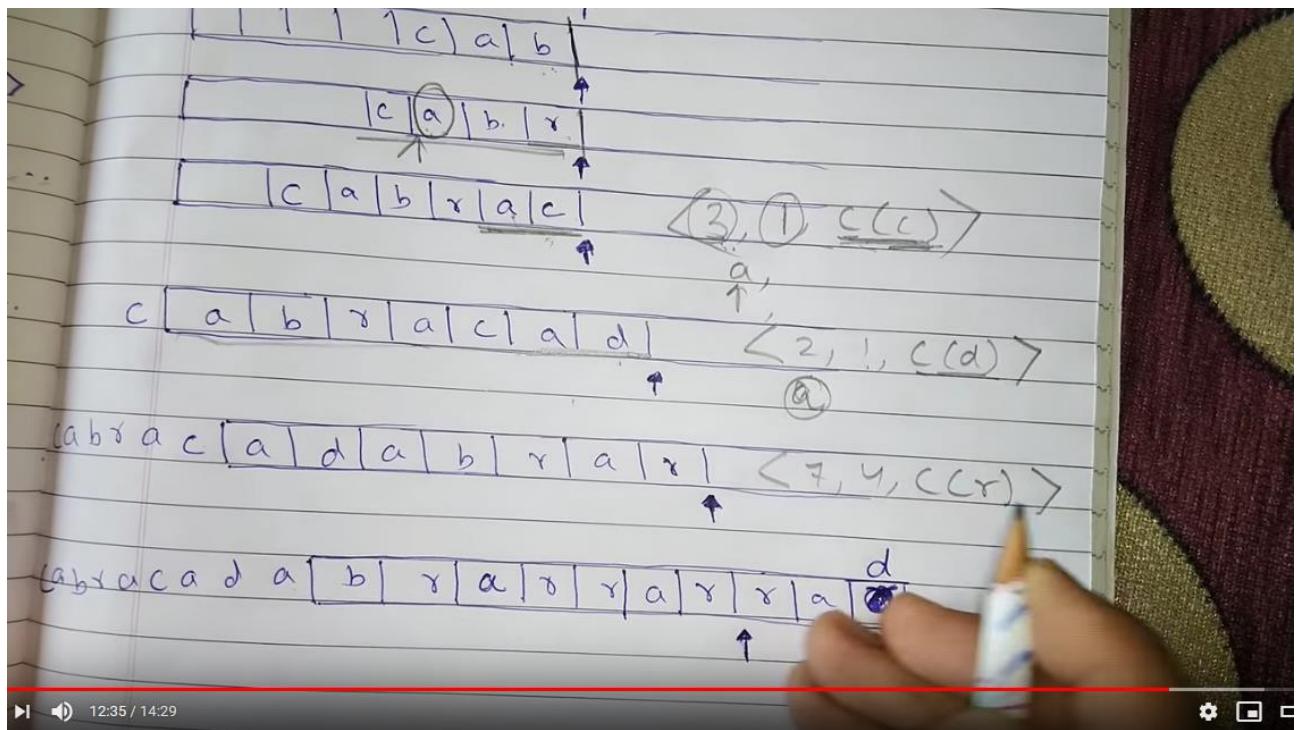
Итогом кодирования будет

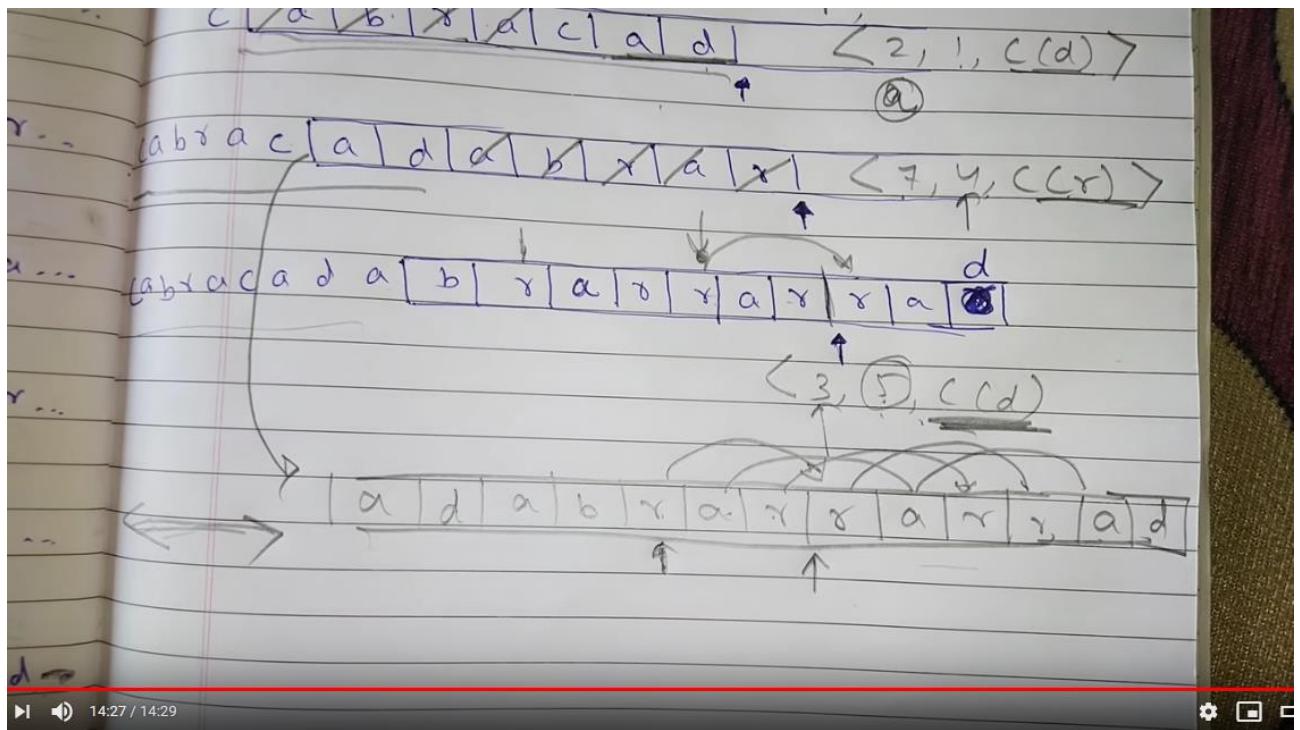
$c \rightarrow \langle 0, 0, c(c) \rangle$   
 $a \rightarrow \langle 0, 0, c(a) \rangle$   
 $b \rightarrow \langle 0, 0, c(b) \rangle$   
 $\gamma \rightarrow \langle 0, 0, c(\gamma) \rangle$   
 $a \rightarrow \langle 3, 1, c(c) \rangle$   
 $a \rightarrow \langle 2, 1, c(b) \rangle$   
 $a \rightarrow \langle 7, 4, c(\gamma) \rangle$   
 $\gamma \rightarrow \langle 3, 5, c(d) \rangle$

#### 14.4.2.2 Декодирование

Используя команды на кодировании раскодируем строку



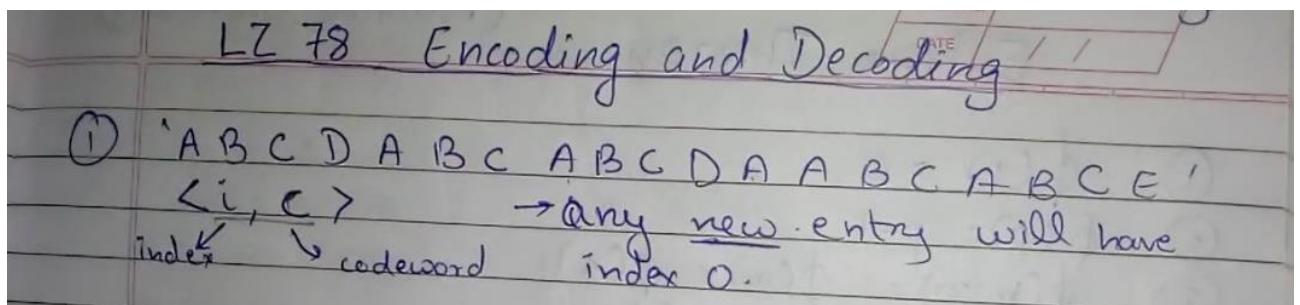




### 14.4.3 LZ78

Алгоритм считывает символы сообщения до тех пор, пока накапливаемая подстрока входит целиком в одну из фраз словаря. Как только эта строка перестанет соответствовать хотя бы одной фразе словаря, алгоритм генерирует код, состоящий из индекса строки в словаре, которая до последнего введенного символа содержала входную строку, и символа, нарушившего совпадение. Затем в словарь добавляется введенная подстрока. Если словарь уже заполнен, то из него предварительно удаляют менее всех используемую в сравнениях фразу.

#### 14.4.3.1 Кодирование



<u>Encoder</u>	<u>Index</u>	<u>codeword</u>	<u>index 0.</u>
$\langle 0, C(A) \rangle$	1		A
$\langle 0, C(B) \rangle$	2		B
$\langle 0, C(C) \rangle$	3		C
$\langle 0, C(D) \rangle$	4		D
$\langle 1, C(B) \rangle$	5		(A)B
$\langle 3, C(A) \rangle$	6		CA
$\langle 2, C(C) \rangle$	7		BC
$\langle 4, C(A) \rangle$	8		DA
$\langle 5, C(C) \rangle$	9		

<u>Encoder</u>	<u>Index</u>	<u>Entry</u>
$\langle 0, C(A) \rangle$	1	A
$\langle 0, C(B) \rangle$	2	B
$\langle 0, C(C) \rangle$	3	C
$\langle 0, C(D) \rangle$	4	D
$\langle 1, C(B) \rangle$	5	(A)B
$\langle 3, C(A) \rangle$	6	CA
$\langle 2, C(C) \rangle$	7	BC
$\langle 4, C(A) \rangle$	8	DA
$\langle 5, C(C) \rangle$	9	(A)BC
$\langle 9, C(E) \rangle$	10	ABCDE

#### 14.4.3.2 Декодирование

<u>Encoder (o/p)</u>	<u>Index</u>	<u>Entry</u>
<0, C(A)>	1	A
<0, C(B)>	2	B
<0, C(C)>	3	C
<0, C(D)>	4	D
<1, C(B)>	5	AB
<3, C(A)>	6	CA
<2, C(C)>	7	BC
<4, C(A)>	8	DA
<5, C(C)>	9	ABC
<9, C(E)>	10	ABCDE

Тоже самое, что и при кодировании, только в конце нужно пройти по всем Encoder последовательно и выписать результат

<u>Encoder (o/p)</u>	<u>Index</u>	<u>Entry</u>
<0, C(A)>	1	A
<0, C(B)>	2	B
<0, C(C)>	3	C
<0, C(D)>	4	D
<1, C(B)>	5	AB
<3, C(A)>	6	CA
<2, C(C)>	7	BC
<4, C(A)>	8	DA
<5, C(C)>	9	ABC
<9, C(E)>	10	ABCDE

A B C D A B C A B C D A A B C A B C G

#### 14.4.4 Сжатие способом кодирования серий

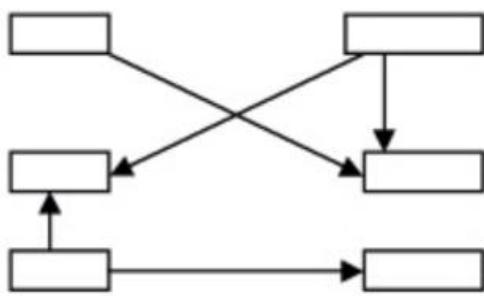
Наиболее известный простой подход и алгоритм сжатия информации обратимым путем – это кодирование серий последовательностей (Run Length Encoding – RLE). Суть методов данного подхода состоит в замене цепочек или

серий повторяющихся байтов или их последовательностей на один кодирующий байт и счетчик числа их повторений. Проблема всех аналогичных методов заключается лишь в определении способа, при помощи которого распаковывающий алгоритм мог бы отличить в результирующем потоке байтов кодированную серию от других – некодированных последовательностей байтов. Решение проблемы достигается обычно простановкой меток в начале кодированных цепочек. Такими метками могут быть, например, характерные значения битов в первом байте кодированной серии, значения первого байта кодированной серии и т.п. Данные методы, как правило, достаточно эффективны для сжатия растровых графических изображений (BMP, PCX, TIF, GIF), т.к. последние содержат достаточно много длинных серий повторяющихся последовательностей байтов. Недостатком метода RLE является достаточно низкая степень сжатия или стоимость кодирования файлов с малым числом серий и, что еще хуже – с малым числом повторяющихся байтов в сериях.

## 15

### 15.1 Реляционная модель

Реляционная модель данных (РМД) — логическая модель данных. Это совокупность отношений, содержащих всю информацию, которая должна храниться в базе.



**Реляционная**  
Таблицы независимы.  
Связи полностью изменчивы.  
Простота расширения.

Термин «реляционный» означает, что теория основана на математическом понятии отношение (relation). В качестве неформального синонима термину «отношение» часто встречается слово таблица.

Результат выполнения любой операции над отношением также является отношением. Эта особенность называется свойством реляционной замкнутости.

Для лучшего понимания РМД следует отметить три важных обстоятельства:

- модель является логической, то есть отношения являются логическими (абстрактными), а не физическими (хранимыми) структурами;
- для реляционных баз данных верен информационный принцип: всё информационное наполнение базы данных представлено одним и только одним способом, а именно — явным заданием значений атрибутов в кортежах отношений; в частности, нет никаких указателей (адресов), связывающих одно значение с другим;
- наличие реляционной алгебры позволяет реализовать декларативное программирование и декларативное описание ограничений целостности, в дополнение к навигационному (процедурному) программированию и процедурной проверке условий.

Кроме того, в состав реляционной модели данных включают теорию нормализации 15.3.

## СУБД / DBMS

→ Дореляционные / Pre-relational

↓ Реляционные / Relational

- Пред-SQL / Pre-SQL  
*(dBase, FoxPro); Paradox*
  - SQL-серверы БД / SQL DB Servers
    - > ◦ Comm: Oracle; DB2; SQL Server;  
SAP ASE (ex: Sybase); Informix; ...
    - Open: (MySQL, MariaDB);  
PostgreSQL; Firebird; ...
  - Встраиваемые / Embedded  
*SQLite; HSQLDB; ЛИНТЕР (Linter);  
Firebird Embedded; ...*
  - Настольные / Desktop  
*Access; OpenOffice Base;  
LibreOffice Base; FileMaker Pro*
- Объектные / Object
- Многомерные / Multidimensional
- NoSQL, NewSQL и др.

## Коммерческие / Commercial SQL-серверы БД / SQL DB Servers

- Oracle Database (Oracle Corp.) — PL/SQL
- DB2 Universal Database (IBM Corp.) — SQL PL,  
PL/SQL
- Microsoft SQL Server (Microsoft Corp.) —  
Transact-SQL (T-SQL)
- SAP ASE (Adaptive Server Enterprise) (SAP SE),  
ex: Sybase SQL Server — Transact-SQL (T-SQL)
- IBM Informix (IBM Corp.) — SPL

## SQL

2-я пол. 70-х — *System R* (IBM, D. Chamberlin)

SQL — Structured Query Language

ex: SEQUEL — Structured English Query Language

Таблица, столбец, строка / Table, Column, Row

Подъязыки: DDL • DML/DQL • DCL • TCL • CCL

SELECT — выборка, рел. алгебра

80-е — «триумфальное шествие» SQL-серверов

*Oracle* (Relation Software Inc., 1979), *DB2* (IBM,

1981), Microsoft/Sybase *SQL Server* (1989), ...

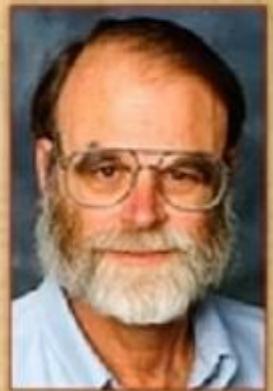
Стандарты: SQL-86, SQL-89, SQL-92, ..., SQL:2008



Дональд  
ЧАМБЕРЛИН  
*Donald D.  
CHAMBERLIN*  
р. 1944



Рэй  
БОЙС  
*Raymond F.  
BOYCE*  
1947–1974



Джим  
ГРЭЙ  
*James Nicholas  
'Jim' GRAY*  
1944–2007 (п.б.)

Реляционная модель ориентирована на организацию данных в виде двумерных таблиц, любая из которых имеет следующие свойства:

- каждый элемент таблицы - это один элемент данных;
- все столбцы в таблицы - однородные, т.е все элементы в столбце имеют одинаковый тип (символьный, числовой и т.п.);
- каждый столбец носит уникальное имя;
- одинаковые строки в таблицы отсутствуют.

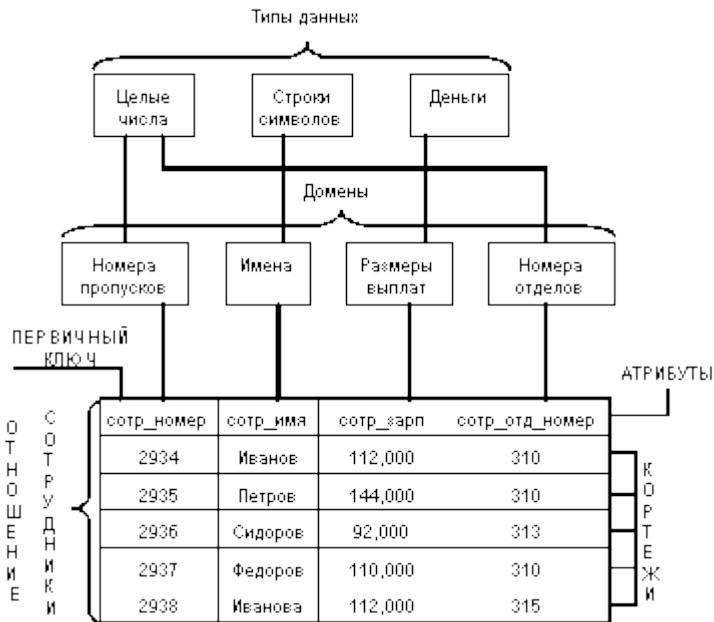
Таблицы имеют строки, которые отвечают записям (или кортежам), а столбцы - атрибутам отношений (доменам, полям).

Следующие термины являются эквивалентными:

- отношение, таблица, файл (для локальных БД);

- кортеж, строка, запись;
- атрибут, столбик, поле.

Покажем смысл этих понятий на примере отношения СОТРУДНИКИ, содержащего информацию о сотрудниках некоторой организации:



Каждое отношение состоит из двух частей — заголовка и тела; выражаясь неформально, заголовок — это атрибуты, а тело — кортежи.

### 15.1.1 Тип данных

Понятие тип данных в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Обычно в современных реляционных БД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (таких как "деньги"), а также специальных "temporalных" данных (дата, время, временной интервал). Достаточно активно развивается подход к расширению возможностей реляционных систем абстрактными типами данных (соответствующими возможностями обладают, например, системы семейства Ingres/Postgres). В нашем примере мы имеем дело с данными трех типов: строки символов, целые числа и "деньги".

### 15.1.2 Атрибут

Атрибут — свойство некоторой сущности. Часто называется полем таблицы.

Каждое отношение состоит из двух частей — заголовка и тела; выражаясь неформально, заголовок — это атрибуты, а тело — кортежи.

### *15.1.3 Домен атрибута*

Множество значений (область определения) атрибута называется доменом.

Домен определяется заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных. Если вычисление этого логического выражения дает результат "истина", то элемент данных является элементом домена.

Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого потенциального множества значений данного типа. Например, домен "Имена" в нашем примере определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут изображать имя (в частности, такие строки не могут начинаться с мягкого знака).

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения доменов "Номера пропусков" и "Номера групп" относятся к типу целых чисел, но не являются сравнимыми. Заметим, что в большинстве реляционных СУБД понятие домена не используется, хотя в Oracle V.7 оно уже поддерживается.

### *15.1.4 Взаимосвязи*

В реляционной БД таблицы взаимосвязаны и соотносятся друг с другом как главные и подчиненные. Связь главной и подчиненной таблицы осуществляется через первичный ключ (primary key) главной таблицы и внешний ключ (foreign key) подчиненной таблицы.

Внешний ключ это атрибут или набор атрибутов, который в главной таблице является первичным ключом.

### *15.1.5 Отношение*

Отношение — конечное множество кортежей (таблица).

### *15.1.6 Кортеж*

Строка в таблице является кортежем.

Конечное упорядоченное множество взаимосвязанных допустимых значений атрибутов, которые вместе описывают некоторую сущность (строка таблицы).

Каждое отношение состоит из двух частей — заголовка и тела; выражаясь неформально, заголовок — это атрибуты, а тело — кортежи.

Кортеж, соответствующий данной схеме отношения, - это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. "Значение" является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень или "арность" кортежа, т.е. число элементов в нем, совпадает с "арностью" соответствующей схемы отношения. Попросту говоря, кортеж - это набор именованных значений заданного типа.

Отношение - это множество кортежей, соответствующих одной схеме отношения. Иногда, чтобы не путаться, говорят "отношение-схема" и "отношение-экземпляр", иногда схему отношения называют заголовком отношения, а отношение как набор кортежей - телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования. Было бы вполне логично разрешать отдельно определять схему отношения, а затем одно или несколько отношений с данной схемой.

Однако в реляционных базах данных это не принято. Имя схемы отношения в таких базах данных всегда совпадает с именем соответствующего отношения-экземпляра. В классических реляционных базах данных после определения схемы базы данных изменяются только отношения-экземпляры. В них могут появляться новые и удаляться или модифицироваться существующие кортежи. Однако во многих реализациях допускается и изменение схемы базы данных: определение новых и изменение существующих схем отношения. Это принято называть эволюцией схемы базы данных.

Обычным житейским представлением отношения является таблица, заголовком которой является схема отношения, а строками - кортежи отношения-экземпляра; в этом случае имена атрибутов именуют столбцы этой таблицы. Поэтому иногда говорят "столбец таблицы", имея в виду "атрибут отношения". Когда мы перейдем к рассмотрению практических вопросов организации реляционных баз данных и средств управления, мы будем использовать эту житейскую терминологию. Этой терминологии придерживаются в большинстве коммерческих реляционных СУБД.

### *15.1.7 Схема отношения*

Схема отношения — конечное множество атрибутов, определяющих некоторую сущность. Иными словами, это структура таблицы, состоящей из конкретного набора полей.

### *15.1.8 Проекция*

Проекция — отношение, полученное из заданного путём удаления и (или) перестановки некоторых атрибутов.

## **15.2 Реляционная алгебра.**

### *15.2.1 Хороший сайт*

[https://function-x.ru/relation\\_algebra.html](https://function-x.ru/relation_algebra.html)

Реляционная алгебра - это язык операций, выполняемых над отношениями - таблицами реляционной базы данных. Операции реляционной алгебры позволяют на основе одного или нескольких отношений создавать другое отношение без изменения самих исходных отношений. Полученное другое отношение обычно не записывается в базу данных, а существует в результате выполнения SQL-запроса - массиве, создаваемом функциями для работы с базами данных в языках программирования. Для каждой операции реляционной алгебры будет дана её реализация в виде запросов на языке SQL.

Рассмотрим операции реляционной алгебры. Чтобы Вам не отвлекаться на содержание таблиц не Ваших баз данных, таких как "Продукты", "Водители", "сливы", "груши", "чай", "кофе", Владимиры, Сергеи и т.п. будем выполнять операции над отношениями (таблицами) с абстрактными данными, такими как R1, R2 (названия таблиц - отношений) и т.д. и A1, A2, A3 (названия атрибутов - столбцов) и h15, w11 и т.п. (содержание записей таблиц базы данных).

Основы реляционной алгебры - математическая теория множеств и операции над множествами. А уйти ещё глубже в теорию реляционных баз данных можно на уроке Реляционная модель данных.

Приоритеты выполнения операций реляционной алгебры (в порядке убывания пунктов списка, а в одном пункте - операции с равными приоритетами):

- селекция, проекция
- декартово произведение, соединение, пересечение, деление
- объединение, разность.

### 15.2.1.1 Операция выборки

Операция выборки работает с одним отношением  $R_1$  и определяет результирующее отношение  $R$ , которое содержит только те кортежи (или строки, или записи), отношения  $R_1$ , которые удовлетворяют заданному условию (предикату  $P$ ).

Таким образом, операция выборки - унарная операция - и записывается следующим образом:

$$R = \sigma_p(R_1),$$

где  $P$  - предикат (логическое условие).

Запрос SQL

```
SELECT * from R3 WHERE A3>'d0'
```

Теперь посмотрим, что получится в результате выполнения этой операции реляционной алгебры и соответствующего ей запроса SQL. В таблице ниже дано одно отношение, с которым работает эта операция.

R3			
A1	A2	A3	A4
3	hh	yl	ms
4	pp	a1	sr
1	rr	yl	ms

Просматриваем столбец A3 и устанавливаем, что предикату  $A3 > 'd0'$  удовлетворяют записи в первой и третьей строках исходного отношения (так как номер буквы у в алфавите больше номера буквы d). В результате получаем следующее новое отношение, в котором две строки:

R			
A1	A2	A3	A4
3	hh	yl	ms
1	rr	yl	ms

### 15.2.1.2 Операция проекции

Операция проекции ( $R = \pi_{a_1, \dots, a_n}(R_1)$ ) работает, как и операция выборки, только с одним отношением  $R_1$  и определяет новое отношение  $R$ , в котором есть лишь те атрибуты (столбцы), которые заданы в операции, и их значения.

Запрос SQL

```
SELECT DISTINCT A4, A3 from R3
```

Пусть вновь дано то же отношение  $R3$ :

R3			
A1	A2	A3	A4
3	hh	yl	ms
4	pp	a1	sr
1	rr	yl	ms

Из исходного отношения выбираем только столбцы  $A4$  и  $A3$  и видим, что строки со значениями - первая и третья - идентичны. Исключаем дубликат (за это отвечает ключевое слово `DISTINCT` в SQL-запросе, которое говорит, что нужно выбрать только уникальные записи) и получаем следующее новое отношение, в котором два атрибута и две строки (записи):

R	
A4	A3
ms	yl
sr	a1

### 15.2.1.3 Операция объединения

Результатом объединения двух множеств (отношений) А и В ( $A \cup B$ ) будет такое множество (отношение) С, которое включает в себя те и только те элементы, которые есть или во множестве А или во множестве В. Говоря упрощённо, все элементы множества А и множества В, за исключением дубликатов, образующихся за счёт того, что некоторые элементы есть и в первом, и во втором множестве. Операция объединения реляционной алгебры идентична операции [объединения множеств, которая также описана в материале "Множества и операции над множествами"](#).

#### Запрос SQL

```
SELECT A1, A2, A3 from R1 UNION SELECT A1, A2, A3 from R2
```

Теперь посмотрим, что получится в результате выполнения этой операции реляционной алгебры и соответствующего ей запроса SQL. Теперь даны два отношения, так как операция объединения - бинарная операция:

R1			R2		
A1	A2	A3	A1	A2	A3
Z7	aa	w11	X8	pp	k21
B7	hh	h15	Q2	ee	h15
X8	pp	w11	X8	pp	w11

Объединяем строки первого и второго отношения и видим, что третья строка, которая является третьей и в первом, и во втором отношении - идентичны, поэтому её включаем в новое отношение только один раз. Получаем следующее отношение:

R		
A1	A2	A3
Z7	aa	w11
B7	hh	h15
X8	pp	w11
X8	pp	k21
Q2	ee	h15

Важно следующее: операция объединения может быть выполнена только тогда, когда два отношения обладают одинаковым числом и названиями атрибутов (столбцов), или, говоря формально, совместимы по объединению.

#### 15.2.1.4 Операция пересечения

Результатом пересечения двух множеств (отношений) A и B ( $A \cap B$ ) будет такое множество (отношение) C, которое включает в себя те и только те элементы, которые есть и во множестве A, и во множестве B. Операция пересечения реляционной алгебры идентична операции [пересечения множеств, которая также описана в материале "Множества и операции над множествами"](#).

##### Запрос SQL

```
SELECT A1, A2, A3 from R1 INTERSECT SELECT A1, A2, A3 from R2
```

В некоторыхialectах SQL отсутствует ключевое слово INTERSECT. Поэтому, например, в MySQL и других, операция пересечения множеств может реализована [с применением предиката EXISTS](#).

Теперь посмотрим, что получится в результате выполнения этой операции реляционной алгебры и соответствующего ей запроса SQL. Вновь даны два отношения R1 и R2:

R1			R2		
A1	A2	A3	A1	A2	A3
Z7	aa	w11	X8	pp	k21
B7	hh	h15	Q2	ee	h15
X8	pp	w11	X8	pp	w11

Просматриваем все записи в двух отношениях, и обнаруживаем, что и в первом, и во втором отношении есть одна строка - та, которая является третьей и в первом, и во втором отношении. Получаем новое отношение:

R		
A1	A2	A3
X8	pp	w11

### 15.2.1.5 Операция разности

Разность двух отношений  $R_1$  и  $R_2$  ( $R_1 - R_2$ ) состоит из кортежей (или записей, или строк), которые имеются в отношении  $R_1$ , но отсутствуют в отношении  $R_2$ . Отношения  $R_1$  и  $R_2$  должны быть совместимы по объединению. Операция разности реляционной алгебры идентична операции [разности множеств, которая также описана в материале "Множества и операции над множествами"](#).

#### Запрос SQL

```
SELECT A1, A2, A3 from R2 EXCEPT  
SELECT A1, A2, A3 from R1
```

Установим, что получится в результате выполнения этой операции реляционной алгебры и соответствующего ей запроса SQL. Вновь даны два отношения  $R_1$  и  $R_2$ :

R1			R2		
A1	A2	A3	A1	A2	A3
Z7	aa	w11	X8	pp	k21
B7	hh	h15	Q2	ee	h15
X8	pp	w11	X8	pp	w11

Из отношения  $R_2$  исключаем строку, которая есть также в отношении  $R_2$  - третью - и получаем новое отношение:

R		
A1	A2	A3
X8	pp	w11
Q2	ee	h15

В некоторых диалектах SQL отсутствует ключевое слово EXCEPT. Поэтому, например, в MySQL и других, операция пересечения множеств может реализована [с применением предиката NOT EXISTS](#).

### 15.2.1.6 Операция декартова произведения

Операция декартова произведения ( $R_1 \times R_2$ ) определяет новое отношение R, которое является результатом конкатенации каждого кортежа отношения R1 с каждым кортежем отношения R2.

#### Запрос SQL

```
SELECT * from R3, R4
```

Установим, что получится в результате выполнения этой операции реляционной алгебры и соответствующего ей запроса SQL. Даны два отношения R3 и R4:

R3				R4	
A1	A2	A3	A4	A5	A6
3	hh	yl	ms	3	hh
4	pp	a1	sr	4	pp
1	rr	yl	ms		

В новом отношении должны присутствовать все атрибуты (столбцы) двух отношений. Сначала первая строка отношения R3 сцепляется с каждой из двух строк отношения R4, затем вторая строка отношения R3, затем третья. В результате должно получиться  $3 \times 2 = 6$  кортежей (строк). Получаем такое новое отношение:

R					
A1	A2	A3	A4	A5	A6
3	hh	yl	ms	3	hh
3	hh	yl	ms	4	pp
4	pp	a1	sr	3	hh
4	pp	a1	sr	4	pp
1	rr	yl	ms	3	hh
1	rr	yl	ms	4	pp

### 15.2.1.7 Операция деления

Результатом операции деления ( $R = R_1 \div R_2$ ) является набор кортежей (строк) отношения  $R_1$ , которые соответствуют комбинации всех кортежей отношения  $R_2$ . Для этого нужно, чтобы в отношении  $R_2$  была часть атрибутов (можно и один), которые есть в отношении  $R_1$ . В результирующем отношении присутствуют только те атрибуты отношения  $R_1$ , которых нет в отношении  $R_2$ .

#### Запрос SQL

```
SELECT DISTINCT A1, A4 from R5 WHERE  
NOT EXIST (SELECT * from R6 WHERE NOT EXIST  
R6.A2 = R5.A2 AND  
R6.A3 = R5.A3)
```

Давайте посмотрим, что получится в результате выполнения этой операции реляционной алгебры и соответствующего ей запроса SQL. Даны два отношения  $R_5$  и  $R_6$ :

R5				R6	
A1	A2	A3	A4	A2	A3
2	S3	4	sun	R4	8
3	X8	7	kab	X8	7
3	R4	8	kab		

Комбинации всех кортежей отношения  $R_6$  соответствуют вторая и третья строки отношения  $R_5$ . Но после исключения атрибутов (столбцов)  $A_2$  и  $A_3$  эти строки становятся идентичными. Поэтому в новом отношении присутствует эта строка один раз. Новое отношение:

R	
A1	A4
3	kab

### 15.2.1.1 Операция соединения

Операция соединения обратна операции проекции и создает новое отношение из двух уже существующих. Новое отношение получается конкатенацией кортежей первого и второго отношений, при этом конкатенации подвергаются отношения, в которых совпадают значения заданных атрибутов.

В частности, если соединить отношения PRODUCTS и SELLERS, этими атрибутами будут атрибуты доменов ID.

Также для понятности можно представить соединение как результат двух операций. Сначала берется произведение исходных таблиц, а потом из

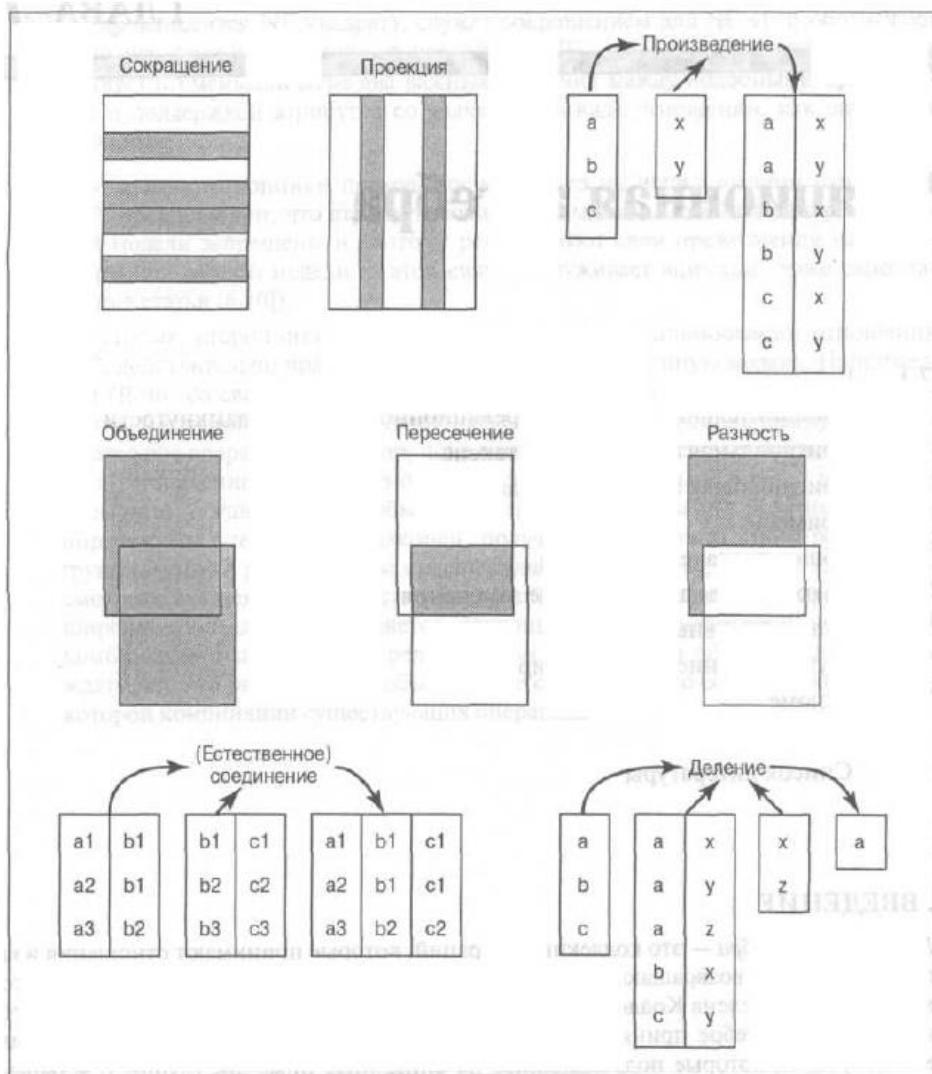
полученного отношения мы делаем выборку с условием равенства атрибутов из одинаковых доменов. В данном случае условием является равенство PRODUCTS.ID и SELLERS.ID (см. 15.2.3)

```
SELECT *
FROM products, sellers
WHERE products.id=sellers.id
```

### 15.2.2 Книга

Реляционная алгебра — это коллекция операций, которые принимают отношения в качестве операндов и возвращают отношение в качестве результата. Первая версия этой алгебры была определена Коддом;. Эта "оригинальная" алгебра включала восемь операций, которые подразделялись на описанные ниже две группы с четырьмя операциями каждая.

- Традиционные операции с множествами — **объединение, пересечение, разность и декартово произведение** (все они были немного модифицированы с учетомтого факта, что их operandами являются именно отношения, а не произвольные множества).
- Специальные реляционные операции, такие как **сокращение** (известное также под названием выборки), **проекция, соединение и деление**.



Прежде всего, отметим, что Кодд, определяя только эти восемь операций, руководствовался весьма конкретным замыслом, как будет описано в следующей главе. Но на этих восьми операциях все не заканчивается; в действительности, может быть определено любое количество операций, которые удовлетворяют простому требованию, чтобы "на входе и выходе были отношения".

Результат выполнения любой операции над отношением также является отношением. Эта особенность называется свойством реляционной замкнутости.

### 15.2.2.1 Объединение

Объединение множеств в теории множеств — множество, содержащее в себе все элементы исходных множеств.

Для БД результатом является множество, состоящее из всех кортежей, присутствующих либо в одном, либо в обоих из заданных отношений. Но хотя этот результат можно назвать множеством, он не является отношением; отношения не могут содержать смесь кортежей разных типов, поскольку они должны включать однотипные кортежи. А нам требуется, чтобы результат

представлял собой отношение, поскольку необходимо сохранить реляционное свойство замкнутости.

Поэтому определение операции реляционного объединения должно быть таким: если даны отношения  $a$  и  $b$  одного и того же типа, то объединение этих отношений  $a \text{ UNION } b$  является отношением того же типа с телом, которое состоит из всех кортежей  $t$ , присутствующих в  $a$  или  $b$  или в обоих отношениях.

A				B			
S#	SNAME	STATUS	CITY	S#	SNAME	STATUS	CITY
S1	Smith	20	London	S1	Smith	20	London
S4	Clark	20	London	S2	Jones	10	Paris

1. Объединение (A UNION B)	<table border="1"> <thead> <tr> <th>S#</th><th>SNAME</th><th>STATUS</th><th>CITY</th></tr> </thead> <tbody> <tr> <td>S1</td><td>Smith</td><td>20</td><td>London</td></tr> <tr> <td>S4</td><td>Clark</td><td>20</td><td>London</td></tr> <tr> <td>S2</td><td>Jones</td><td>10</td><td>Paris</td></tr> </tbody> </table>	S#	SNAME	STATUS	CITY	S1	Smith	20	London	S4	Clark	20	London	S2	Jones	10	Paris
S#	SNAME	STATUS	CITY														
S1	Smith	20	London														
S4	Clark	20	London														
S2	Jones	10	Paris														
2. Пересечение (A INTERSECT B)	<table border="1"> <thead> <tr> <th>S#</th><th>SNAME</th><th>STATUS</th><th>CITY</th></tr> </thead> <tbody> <tr> <td>S1</td><td>Smith</td><td>20</td><td>London</td></tr> </tbody> </table>	S#	SNAME	STATUS	CITY	S1	Smith	20	London								
S#	SNAME	STATUS	CITY														
S1	Smith	20	London														
3. Разность (A MINUS B)	<table border="1"> <thead> <tr> <th>S#</th><th>SNAME</th><th>STATUS</th><th>CITY</th></tr> </thead> <tbody> <tr> <td>S4</td><td>Clark</td><td>20</td><td>London</td></tr> </tbody> </table>	S#	SNAME	STATUS	CITY	S4	Clark	20	London								
S#	SNAME	STATUS	CITY														
S4	Clark	20	London														
4. Разность (B MINUS A)	<table border="1"> <thead> <tr> <th>S#</th><th>SNAME</th><th>STATUS</th><th>CITY</th></tr> </thead> <tbody> <tr> <td>S2</td><td>Jones</td><td>10</td><td>Paris</td></tr> </tbody> </table>	S#	SNAME	STATUS	CITY	S2	Jones	10	Paris								
S#	SNAME	STATUS	CITY														
S2	Jones	10	Paris														

### 15.2.2.2 Пересечение

Результатом операции пересечения будет отношение, состоящее из кортежей, полностью входящих в состав обоих отношений.

Как и для объединения, и фактически по той же причине, для реляционной операции пересечения требуется, чтобы ее operandы принадлежали к одному и тому же типу. Если даны отношения  $a$  и  $b$  одного и того же типа, то пересечением этих отношений  $a \text{ INTERSECT } b$  является отношение того же типа с телом, состоящим из всех кортежей  $t$ , таких, что  $t$  присутствует одновременно в  $a$  и  $b$ .

См пример в 15.2.2.1.

### 15.2.2.3 Разность

Разность двух отношений  $R_1$  и  $R_2$  ( $R_1 - R_2$ ) состоит из кортежей (или записей, или строк), которые имеются в отношении  $R_1$ , но отсутствуют в отношении  $R_2$ . Отношения  $R_1$  и  $R_2$  должны быть совместимы по объединению. Операция разности реляционной алгебры идентична операции разности множеств.

```
SELECT A1, A2, A3 from R2 EXCEPT  
SELECT A1, A2, A3 from R1
```

Установим, что получится в результате выполнения этой операции реляционной алгебры и соответствующего ей запроса SQL. Вновь даны два отношения  $R_1$  и  $R_2$ :

R1			R2		
A1	A2	A3	A1	A2	A3
Z7	aa	w11	X8	pp	k21
B7	hh	h15	Q2	ee	h15
X8	pp	w11	X8	pp	w11

Из отношения  $R_2$  исключаем строку, которая есть также в отношении  $R_2$  - третью - и получаем новое отношение:

R		
A1	A2	A3
X8	pp	w11
Q2	ee	h15

В некоторых диалектах SQL отсутствует ключевое слово EXCEPT. Поэтому, например, в MySQL и других, операция пересечения множеств может реализована с применением предиката NOT EXISTS.

Как и для объединения и пересечения, для реляционной операции разности требуется, чтобы ее operandы принадлежали к одному и тому же типу. Если даны отношения  $a$  и  $b$  одного и того же типа, то разностью этих отношений  $a \text{ MINUS } b$  (в указанном порядке), является отношение того же типа с телом, состоящим из всех кортежей  $t$ , таких, что  $t$  присутствует в  $a$ , но не в  $b$ .

#### *15.2.2.4 Произведение*

В математике декартовым произведением (или сокращенно произведением) двух множеств является множество всех таких упорядоченных пар, что в каждой паре первый элемент берется из первого множества, а второй — из второго множества. Поэтому декартово произведение двух отношений, неформально выражаясь, представляет собой множество упорядоченных пар кортежей. Но мы снова должны сохранить свойство замкнутости; иными словами, необходимо, чтобы результат содержал кортежи как таковые, а не упорядоченные пары кортежей. Поэтому реляционной версией декартова произведения служит расширенная форма этой операции, в которой каждая упорядоченная пара кортежей заменяется одним кортежем, являющимся объединением двух рассматриваемых кортежей (в данном случае термин объединение используется в своем обычном смысле теории множеств, а не в его особом реляционном смысле). Это означает, что если даны следующие кортежи

$$\{ A_1 a_1, A_2 a_2, \dots, A_m a_m \}$$

И

$$\{ B_1 b_1, B_2 b_2, \dots, B_n b_n \}$$

то теоретико-множественное объединение этих двух кортежей представляет собой приведенный ниже единственный кортеж.

$$\{ A_1 a_1, A_2 a_2, \dots, A_m a_m, B_1 b_1, B_2 b_2, \dots, B_n b_n \}$$

Пример. Предположим, что отношения А и В показаны на рис. 7.3 (неформально выражаясь, отношение А содержит все текущие номера поставщиков, а в — все текущие номера деталей). В таком случае декартово произведение А TIMES В (которое показано в нижней части данного рисунка) включает в себя все текущие пары номеров поставщиков и номеров деталей.

A	S #	B	P #
	S1		P1
	S2		P2
	S3		P3
	S4		P4
	S5		P5
			P6

Декартово произведение (A TIMES B)

S#	P#	..	..	..	..	..	..	..	..	..
S1	P1	S2	P1	S3	P1	S4	P1	S5	P1	..
S1	P2	S2	P2	S3	P2	S4	P2	S5	P2	..
S1	P3	S2	P3	S3	P3	S4	P3	S5	P3	..
S1	P4	S2	P4	S3	P4	S4	P4	S5	P4	..
S1	P5	S2	P5	S3	P5	S4	P5	S5	P5	..
S1	P6	S2	P6	S3	P6	S4	P6	S5	P6	..
..	..	..	..	..	..	..	..	..	..	..

### 15.2.2.5 Сокращение (Выборка, Ограничение)

Сокращение — это операция, которая выделяет множество строк в таблице, удовлетворяющих заданным условиям. Условием может быть любое логическое выражение результат которого - ИСТИНА.

$\sigma_{(PRICE > 90)}$  PRODUCTS

SELECT \* FROM products WHERE price>90;

### 15.2.2.6 Проекция

Проекция является операцией, при которой из отношения выделяются атрибуты только из указанных доменов, то есть из таблицы выбираются только нужные столбцы, при этом, если получится несколько одинаковых кортежей, то в результирующем отношении остается только по одному экземпляру подобного кортежа.

$\pi_{(ID, PRICE)}$  PRODUCTS

SELECT id, price FROM products;

На практике часто удобно иметь возможность указывать не атрибуты, по которым должна быть сформирована проекция, а скорее те атрибуты, которые

"должны быть отброшены" (т.е. удалены) после выполнения операции проекции, однако в СУБД такой подход не реализован, тк

- Это делает ваш контракт между клиентом и базой данных стабильным.
- Те же данные, каждый раз
- Производительность, охватывающая индексы

### *15.2.2.7 Соединение*

Операция соединения обратна операции проекции и создает новое отношение из двух уже существующих. Новое отношение получается конкатенацией кортежей первого и второго отношений, при этом конкатенации подвергаются отношения, в которых совпадают значения заданных атрибутов.

В частности, если соединить отношения PRODUCTS и SELLERS, этими атрибутами будут атрибуты доменов ID.

Также для понятности можно представить соединение как результат двух операций. Сначала берется произведение исходных таблиц, а потом из полученного отношения мы делаем выборку с условием равенства атрибутов из одинаковых доменов. В данном случае условием является равенство PRODUCTS.ID и SELLERS.ID (см. 15.2.2.8)

```
SELECT *
FROM products, sellers
WHERE products.id=sellers.id
```

### *15.2.2.8 Деление*

Результатом операции деления ( $R = R_1 \div R_2$ ) является набор кортежей (строк) отношения R1, которые соответствуют комбинации всех кортежей отношения R2. Для этого нужно, чтобы в отношении R2 была часть атрибутов (можно и один), которые есть в отношении R1. В результирующем отношении присутствуют только те атрибуты отношения R1, которых нет в отношении R2.

```
SELECT DISTINCT A1, A4 from R5 WHERE
NOT EXIST (SELECT * from R6 WHERE NOT EXIST
R6.A2 = R5.A2 AND
R6.A3 = R5.A3)
```

Давайте посмотрим, что получится в результате выполнения этой операции реляционной алгебры и соответствующего ей запроса SQL. Даны два отношения R5 и R6:

R5					R6	
A1	A2	A3	A4		A2	A3
2	S3	4	sun		R4	8
3	X8	7	kab		X8	7
3	R4	8	kab			

Комбинации всех кортежей отношения R6 соответствуют вторая и третья строки отношения R5. Но после исключения атрибутов (столбцов) A2 и A3 эти строки становятся идентичными. Поэтому в новом отношении присутствует эта строка один раз. Новое отношение:

R	
A1	A4
3	kab

### 15.2.3 Далее из веба, другое объяснение

Реляционная алгебра представляет собой набор таких операций над отношениями, что результат каждой из операций также является отношением. Это свойство алгебры называется замкнутостью.

Операции над одним отношением называются унарными, над двумя отношениями — бинарными.

Для примера будем использовать следующие таблицы

таблица PRODUCTS

ID	NAME	COMPANY	PRICE
123	Печеньки	ООО "Темная сторона"	190
156	Чай	ООО "Темная сторона"	60
235	Ананасы	ОАО "Фрукты"	100
623	Томаты	ООО "Овощи"	130

---

таблица DRIVERS

COMPANY	DRIVER
ООО "Темная сторона"	Владимир
ООО "Темная сторона"	Михаил
ОАО "Фрукты"	Руслан
ООО "Овощи"	Владимир

---

таблица SELLERS

ID	SELLER
123	ООО "Дарт"
156	ОАО "Ведро"
235	ЗАО "Овоще База"
623	ОАО "Фирма"

Условимся, что в этой таблице ID это внешний ключ, связанный с первичным ключом таблицы PRODUCTS.

SQL запрос создания БД на Postgres лежит по пути */sql/15.2.sql*

### *15.2.3.1 Проекция*

Проекция является операцией, при которой из отношения выделяются атрибуты только из указанных доменов, то есть из таблицы выбираются только нужные столбцы, при этом, если получится несколько одинаковых кортежей, то в результирующем отношении остается только по одному экземпляру подобного кортежа.

Для примера сделаем проекцию на таблице PRODUCTS выбрав из нее ID и PRICE. В результате этой операции получим отношение:

ID	PRICE
123	190
156	60
235	100
623	130

## $\pi_{(ID, PRICE)}$ PRODUCTS

`SELECT id, price FROM products;`

### 15.2.3.2 Выборка

Выборка — это операция, которая выделяет множество строк в таблице, удовлетворяющих заданным условиям. Условием может быть любое логическое выражение.

Для примера сделаем выборку из таблицы с ценой больше 90.

ID	NAME	COMPANY	PRICE
123	Печеньки	ООО "Темная сторона"	190
235	Ананасы	ОАО "Фрукты"	100
623	Томаты	ООО "Овощи"	130

## $\sigma_{(PRICE>90)}$ PRODUCTS

`SELECT * FROM products WHERE price>90;`

В условии выборки мы можем использовать любое логическое выражение.

Сделаем еще одну выборку с ценой больше 90 и ID товара меньше 300:

ID	NAME	COMPANY	PRICE
123	Печеньки	ООО "Темная сторона"	190
235	Ананасы	ОАО "Фрукты"	100

## $\sigma_{(PRICE>90 \wedge ID<300)}$ PRODUCTS

`SELECT * FROM PRODUCTS WHERE PRICE>90 AND ID<300;`

Совместим операторы проекции и выборки. Мы можем это сделать, потому что любой из операторов в результате возвращает отношение и в качестве аргументов использует также отношение.

Из таблицы с продуктами выберем все компании, продающие продукты дешевле 110.

COMPANY
ООО "Темная сторона"
ОАО "Фрукты"

$\pi_{\text{COMPANY}} \sigma_{(\text{PRICE} < 100)} \text{PRODUCTS}$

```
SELECT company FROM products WHERE price<110;
```

### 15.2.3.3 Умножение (Декартово произведение)

Умножение или декартово произведение является операцией, производимой над двумя отношениями, в результате которой мы получаем отношение со всеми доменами из двух начальных отношений. Кортежи в этих доменах будут представлять из себя все возможные сочетания кортежей из начальных отношений. На примере будет понятнее.

Получим декартово произведения таблиц PRODUCTS и SELLERS.

Синтаксис операции:

PRODUCTS  $\times$  SELLERS

```
SELECT *
FROM products, sellers
WHERE products.id<235 AND
      sellers.id<235
```

Можно заметить, что у двух этих таблиц есть одинаковый домен ID. В подобной ситуации домены с одинаковыми названиями получают префикс в виде названия соответствующего отношения, как показано ниже.

Для краткости перемножим не полные отношения, а выборки с условием ID<235. Цветом выделены одни и те же кортежи

PRODUCTS.ID	NAME	COMPANY	PRICE	SELLERS.ID	SELLER
123	Печеньки	ООО "Темная сторона"	190	123	ООО "Дарт"
156	Чай	ООО "Темная сторона"	60	156	ОАО "Ведро"
123	Печеньки	ООО "Темная сторона"	190	156	ОАО "Ведро"
156	Чай	ООО "Темная сторона"	60	123	ООО "Дарт"

Для примера использования этой операции представим себе необходимость выбрать продавцов с ценами меньше 90. Без произведения необходимо было бы сначала получить ID продуктов из первой таблицы, потом по этим ID из второй таблицы получить нужные имена SELLER, а с использованием произведения будет такой запрос:

$$\pi_{(SELLER)} \sigma_{(RODUCTS.ID=SELLERS.ID \wedge PRICE < 90)} PRODUCTS \times SELLERS$$

```

SELECT seller
FROM products, sellers
WHERE products.id=sellers.id AND
price<90

```

SELLER

ОАО "Ведро"

#### 15.2.3.4 Соединение

Операция соединения обратна операции проекции и создает новое отношение из двух уже существующих. Новое отношение получается конкатенацией кортежей первого и второго отношений, при этом конкатенации подвергаются отношения, в которых совпадают значения заданных атрибутов. В частности, если соединить отношения PRODUCTS и SELLERS, этими атрибутами будут атрибуты доменов ID.

Также для понятности можно представить соединение как результат двух операций. Сначала берется произведение исходных таблиц, а потом из полученного отношения мы делаем выборку с условием равенства атрибутов из

одинаковых доменов. В данном случае условием является равенство PRODUCTS.ID и SELLERS.ID.

Попробуем соединить отношения PRODUCTS и SELLERS и получим отношение.

PRODUCTS.ID	NAME	COMPANY	PRICE	SELLERS.ID	SELLER
123	Печеньки	ООО "Темная сторона"	190	123	ООО "Дарт"
156	Чай	ООО "Темная сторона"	60	156	ОАО "Ведро"
235	Ананасы	ОАО "Фрукты"	100	235	ЗАО "Овоще База"
623	Томаты	ООО "Овощи"	130	623	ОАО "Фирма"

```
SELECT *
FROM products, sellers
WHERE products.id=sellers.id
```

Натуральное соединение получает схожее отношение, но в случае, если у нас корректно настроена схема в базе ( в данном случае первый ключ таблицы PRODUCTS ID связан с внешним ключем таблицы SELLERS ID), то в результирующем отношении остается один домен ID.

Синтаксис операции:

PRODUCTS  $\bowtie$  SELLERS;

```
CREATE TABLE sellers2 (
    id      integer PRIMARY KEY NOT NULL AUTO_INCREMENT,
    product integer references products(id),
    seller  varchar(50) NOT NULL
);
```

```
SELECT * FROM products
INNER JOIN sellers2 ON products.id=sellers2.product
```

PRODUCTS.ID	NAME	COMPANY	PRICE	SELLER
123	Печеньки	ООО "Темная сторона"	190	ООО "Дарт"
156	Чай	ООО "Темная сторона"	60	ОАО "Ведро"
235	Ананасы	ОАО "Фрукты"	100	ЗАО "Овоще База"
623	Томаты	ООО "Овощи"	130	ОАО "Фирма"

#### *15.2.3.5 Пересечение и вычитание.*

Результатом операции пересечения будет отношение, состоящее из кортежей, полностью входящих в состав обоих отношений.

Результатом вычитания будет отношение, состоящее из кортежей, которые являются кортежами первого отношения и не являются кортежами второго отношения.

### **15.3 Нормальные формы отношений**

**Нормальная форма** — требование, предъявляемое к структуре таблиц в теории реляционных баз данных для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц).

Метод нормальных форм (НФ) состоит в сборе информации о объектах решения задачи в рамках одного отношения и последующей декомпозиции этого отношения на несколько взаимосвязанных отношений на основе процедур нормализации отношений.

**Цель нормализации:** исключить избыточное дублирование данных, которое является причиной аномалий, возникших при добавлении, редактировании и удалении кортежей(строк таблицы).

Процесс проектирования БД с использованием метода НФ является итерационным и заключается в последовательном переводе отношения из 1НФ в НФ более высокого порядка по определенным правилам. Каждая следующая НФ ограничивается определенным типом функциональных зависимостей и устранением соответствующих аномалий при выполнении операций над отношениями БД, а также сохранении свойств предшествующих НФ.

Прежде чем приступить к рассмотрению процедуры нормализации, следует обсудить один существенный аспект этой процедуры, а именно — концепцию декомпозиции без потерь. Как уже упоминалось, процедура

нормализации предусматривает разбиение, или декомпозицию, данной переменной отношения на другие переменные отношения, причем декомпозиция должна быть обратимой, т.е. выполняться без потерь информации.

### 15.3.1 Термины

**Аномалией** называется такая ситуация в таблице БД, которая приводит к противоречию в БД либо существенно усложняет обработку БД. Причиной является излишнее дублирование данных в таблице, которое вызывается наличием функциональных зависимостей от не ключевых атрибутов.

Аномалии-модификации проявляются в том, что изменение одних данных может повлечь просмотр всей таблицы и соответствующее изменение некоторых записей таблицы.

Аномалии-удаления — при удалении какого-либо кортежа из таблицы может пропасть информация, которая не связана на прямую с удаляемой записью.

Аномалии-добавления возникают, когда информацию в таблицу нельзя поместить, пока она не полная, либо вставка записи требует дополнительного просмотра таблицы.

Функциональная зависимость между атрибутами (множествами атрибутов) X и Y означает, что для любого допустимого набора кортежей в данном отношении: если два кортежа совпадают по значению X, то они совпадают по значению Y. Например, если значение атрибута «Название компании» — Canonical Ltd, то значением атрибута «Штаб-квартира» в таком кортеже всегда будет Millbank Tower, London, United Kingdom. Обозначение:  $\{X\} \rightarrow \{Y\}$ .

**Транзитивность** — свойство бинарного отношения. Бинарное отношение  $R$  на множестве  $X$  называется *транзитивным*, если для любых трёх элементов множества  $a, b, c$  выполнение отношений  $aRb$  и  $bRc$  влечёт выполнение отношения  $aRc$ .

### 15.3.2 Первая нормальная форма

Отношение находится в 1НФ, если все его атрибуты являются простыми (не массивы, листы и пр. Пример простого - Фамилия, пример составного - ФИО), все используемые домены должны содержать только скалярные значения. Не должно быть повторений строк в таблице.

Например, есть таблица «Автомобили»:

Фирма	Модели
BMW	M5, X5M, M1
Nissan	GT-R

Нарушение нормализации 1НФ происходит в моделях BMW, т.к. в одной ячейке содержится список из 3 элементов: M5, X5M, M1, т.е. он не является атомарным. Преобразуем таблицу к 1НФ:

Фирма	Модели
BMW	M5
BMW	X5M
BMW	M1
Nissan	GT-R

### 15.3.3 Вторая нормальная форма

Отношение находится во 2НФ, если оно находится в 1НФ и каждый не ключевой атрибут неприводимо зависит от Первичного Ключа(ПК).

Неприводимость означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость.

Например, дана таблица:

Модель	Фирма	Цена	Скидка
M5	BMW	5500000	5%
X5M	BMW	6000000	5%
M1	BMW	2500000	5%
GT-R	Nissan	5000000	10%

Таблица находится в первой нормальной форме, но не во второй. Цена машины зависит от модели и фирмы. Скидка зависит от фирмы, то есть зависимость от первичного ключа неполная. Исправляется это путем декомпозиции на два отношения, в которых не ключевые атрибуты зависят от ПК (первичного ключа).

<u>Модель</u>	<u>Фирма</u>	<u>Цена</u>
M5	BMW	5500000
X5M	BMW	6000000
M1	BMW	2500000
GT-R	Nissan	5000000

<u>Фирма</u>	<u>Скидка</u>
BMW	5%
Nissan	10%

#### *15.3.4 Третья нормальная форма*

Отношение находится в ЗНФ, когда находится во 2НФ и каждый не ключевой атрибут нетранзитивно зависит от первичного ключа. Проще говоря, второе правило требует выносить все не ключевые поля, содержимое которых может относиться к нескольким записям таблицы в отдельные таблицы.

Рассмотрим таблицу:

<u>Модель</u>	<u>Магазин</u>	<u>Телефон</u>
BMW	Риал-авто	87-33-98
Audi	Риал-авто	87-33-98
Nissan	Некст-Авто	94-54-12

Таблица находится во 2НФ, но не в ЗНФ.

В отношении атрибут «Модель» является первичным ключом. Личных телефонов у автомобилей нет, и телефон зависит исключительно от магазина.

Таким образом, в отношении существуют следующие функциональные зависимости: Модель → Магазин, Магазин → Телефон, Модель → Телефон.

Зависимость Модель → Телефон является транзитивной, следовательно, отношение не находится в ЗНФ.

В результате разделения исходного отношения получаются два отношения, находящиеся в ЗНФ:

<u>Магазин</u>	<u>Телефон</u>
Риал-авто	87-33-98
Некст-Авто	94-54-12

<u>Модель</u>	<u>Магазин</u>
BMW	Риал-авто
Audi	Риал-авто
Nissan	Некст-Авто

### *15.3.5 Нормальная форма Бойса-Кодда (НФБК) (частная форма третьей нормальной формы)*

Определение ЗНФ не совсем подходит для следующих отношений:

- 1) отношение имеет два или более потенциальных ключа;
- 2) два и более потенциальных ключа являются составными;
- 3) они пересекаются, т.е. имеют хотя бы один атрибут.

Для отношений, имеющих один потенциальный ключ (первичный), НФБК является ЗНФ.

Отношение находится в НФБК, когда каждая нетривиальная и неприводимая слева функциональная зависимость обладает потенциальным ключом в качестве детерминанта.

Предположим, рассматривается отношение, представляющее данные о бронировании стоянки на день:

Номер стоянки	Время начала	Время окончания	Тариф
1	09:30	10:30	Бережливый
1	11:00	12:00	Бережливый
1	14:00	15:30	Стандарт
2	10:00	12:00	Премиум-В
2	12:00	14:00	Премиум-В
2	15:00	18:00	Премиум-А

Тариф имеет уникальное название и зависит от выбранной стоянки и наличия льгот, в частности:

- «Бережливый»: стоянка 1 для льготников
- «Стандарт»: стоянка 1 для не льготников
- «Премиум-А»: стоянка 2 для льготников
- «Премиум-В»: стоянка 2 для не льготников.

Таким образом, возможны следующие составные первичные ключи:

- {Номер стоянки, Время начала},
- {Номер стоянки, Время окончания},
- {Тариф, Время начала},
- {Тариф, Время окончания}.

Отношение находится в ЗНФ. Требования второй нормальной формы выполняются, так как все атрибуты входят в какой-то из потенциальных ключей, а неключевых атрибутов в отношении нет. Также нет и транзитивных зависимостей, что соответствует требованиям третьей нормальной формы. Тем не менее, существует функциональная зависимость Тариф → Номер стоянки, в которой левая часть (детерминант) не является потенциальным ключом отношения, то есть отношение не находится в нормальной форме Бойса — Кодда.

Недостатком данной структуры является то, что, например, по ошибке можно приписать тариф «Бережливый» к бронированию второй стоянки, хотя он может относиться только к первой стоянки.

Можно улучшить структуру с помощью декомпозиции отношения на два и добавления атрибута Имеет льготы, получив отношения, удовлетворяющие НФБК (подчёркнуты атрибуты, входящие в первичный ключ.):

### **Тарифы**

<u>Тариф</u>	Номер стоянки	Имеет льготы
Бережливый	1	Да
Стандарт	1	Нет
Премиум-А	2	Да
Премиум-В	2	Нет

### **Бронирование**

<u>Тариф</u>	<u>Время начала</u>	<u>Время окончания</u>
Бережливый	09:30	10:30
Бережливый	11:00	12:00
Стандарт	14:00	15:30
Премиум-В	10:00	12:00
Премиум-В	12:00	14:00
Премиум-А	15:00	18:00

#### *15.3.6 Четвертая нормальная форма*

Отношение находится в 4НФ, если оно находится в НФБК и все нетривиальные многозначные зависимости фактически являются функциональными зависимостями от ее потенциальных ключей.

В отношении R (A, B, C) существует **многозначная зависимость**  $R.A \rightarrow -R.B$  в том и только в том случае, если множество значений B, соответствующее паре значений A и C, зависит только от A и не зависит от C.

Предположим, что рестораны производят разные виды пиццы, а службы доставки ресторанов работают только в определенных районах города. Составной первичный ключ соответствующей переменной отношения включает три атрибута: {Ресторан, Вид пиццы, Район доставки}.

Такая переменная отношения не соответствует 4НФ, так как существует следующая многозначная зависимость:

$$\{\text{Ресторан}\} \rightarrow \{\text{Вид пиццы}\}$$

$$\{\text{Ресторан}\} \rightarrow \{\text{Район доставки}\}$$

То есть, например, при добавлении нового вида пиццы придется внести по одному новому кортежу для каждого района доставки. Возможна логическая аномалия, при которой определенному виду пиццы будут соответствовать лишь некоторые районы доставки из обслуживаемых рестораном районов.

Для предотвращения аномалии нужно декомпозировать отношение, разместив независимые факты в разных отношениях. В данном примере следует выполнить декомпозицию на {Ресторан, Вид пиццы} и {Ресторан, Район доставки}.

Однако, если к исходной переменной отношения добавить атрибут, функционально зависящий от потенциального ключа, например цену с учётом стоимости доставки ( $\{\text{Ресторан}, \text{Вид пиццы}, \text{Район доставки}\} \rightarrow \text{Цена}$ ), то полученное отношение будет находиться в 4НФ и его уже нельзя подвергнуть декомпозиции без потерь.

### *15.3.7 Пятая нормальная форма*

Отношения находятся в 5НФ, если оно находится в 4НФ и отсутствуют сложные зависимые соединения между атрибутами.

Если «Атрибут\_1» зависит от «Атрибута\_2», а «Атрибут\_2» в свою очередь зависит от «Атрибута\_3», а «Атрибут\_3» зависит от «Атрибута\_1», то все три атрибута обязательно входят в один кортеж.

Это очень жесткое требование, которое можно выполнить лишь при дополнительных условиях. На практике трудно найти пример реализации этого требования в чистом виде.

Этого на практике не бывает. Покупатель свободен в своем выборе товаров. Поэтому для устранения отмеченного затруднения все три атрибута разносят по разным отношениям (таблицам). После выделения трех новых отношений (Поставщик, Товар и Покупатель) необходимо помнить, что при извлечении информации (например, о покупателях и товарах) необходимо в

запросе соединить все три отношения. Любая комбинация соединения двух отношений из трех неминуемо приведет к извлечению неверной (некорректной) информации. Некоторые СУБД снабжены специальными механизмами, устраняющими извлечение недостоверной информации. Тем не менее, следует придерживаться общей рекомендации: структуру базы данных строить таким образом, чтобы избежать применения 4НФ и 5НФ.

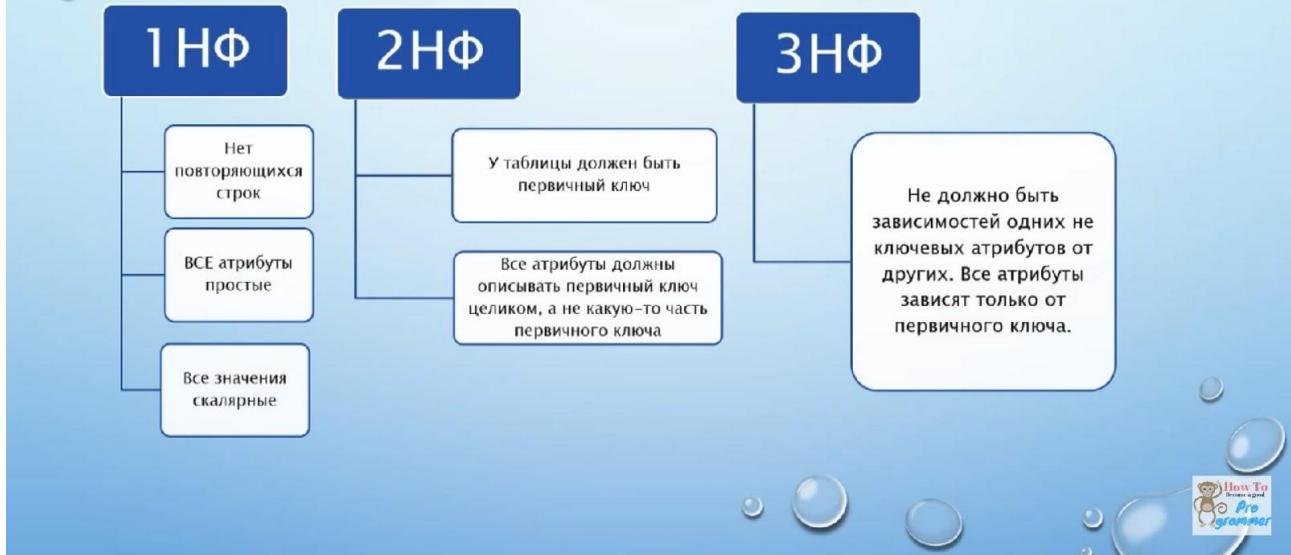
Пятая нормальная форма ориентирована на работу с зависимыми соединениями. Указанные зависимые соединения между тремя атрибутами встречаются очень редко. Зависимые соединения между четырьмя, пятью и более атрибутами указать практически невозможно.

### *15.3.8 Размышления из ютуба*

Только пытаюсь вникать, но вижу что из раза в раз таблицы пытаются упростить и разделить на несколько. Получается в итоге что, идеальный вариант это таблица всего их 2х столбцов, уникальный ключ и одно простое значение к нему? Потому что проще уже наверное не сделать, ибо таблица состоящая только из одного ключа смысла не имеет. Зачем тогда все это было придумывать, все эти формы. Ну будет у меня не 10 таблиц с 5 столбцами, а 100 с двумя.

Это очень правильный вопрос. На самом деле, вы немного переоцениваете значимость нормализации =). Но суть вы уловили абсолютно верно: чем выше нормальная форма, тем меньше дублирующихся данных, но в то же время - тем больше таблиц и связей между ними (ключей). Если все максимально нормализовано в базе данных, то будет огромное количество таблиц. А следовательно - очень тормозная база, поскольку, чтобы что-то путевое из нее выбрать, нужно будет все необходимые таблицы собрать в одну через JOIN (а это трудозатратное действие и хорошенко отъедает производительность). Идеально - это не шестая нормальная форма, и не пятая, и вообще никакая другая НФ. Нет идеальных НФ! =). Идеально - это когда вам удалось соблюсти баланс между отсутствием дублирующихся данных и приемлемой производительностью. Главное - научитесь понимать, как должна выглядеть нормальная таблица, чтобы она не приносила проблем в процессе эксплуатации. Остальное - это лирика и академические изыски. Поначалу НФ и вся эта теория вам здорово помогут разобраться на примерах, но потом все это забудется и останется в сухом остатке опыта, который позволит интуитивно определять, какую таблицу нужно разбивать, а какую нет (конечно, если продолжать заниматься базами).

## 3-Я НОРМАЛЬНАЯ ФОРМА



## 16

### 16.1 Структуры физического уровня баз данных.

#### 16.1.1 Файлы

Основными элементами структуры являются:

- Файлы данных, которые содержат собственно данные и различные объекты, входящие в логическую структуру базы данных, такие как триггеры, хранимые процедуры и т.д.
- Файлы журнала транзакций содержат сведения о ходе выполнения транзакций. В них содержится информация о том, когда началась и закончилась транзакция, какие ресурсы, кем и когда были заблокированы или разблокированы и т.д. См 17.3.1.

Основной или главный файл (Primary File) – присутствует в любой базе данных. Если в базе данных всего один файл с данными, то он является главным. Главный файл содержит так называемые метаданные – данные о том, как устроена база данных (системные таблицы...). Кроме системной информации, главный файл может хранить и пользовательские данные. Главный файл имеет расширение mdf (Master Data File).

Вторичный или дополнительный файл (Secondary File) – дополнительный файл для хранения пользовательских данных. Не содержит системной

информации. Такие файлы могут создаваться администратором по мере необходимости. Имеет расширение `ndf` (secOndary Data File).

Файлы журнала транзакций бывают только одного типа – файлы журнала транзакций (`Transaction Log File`) , расширение `ldf` (Log Data File).

### *16.1.2 Страницы*

Для работы с файлами в активно применяется принцип постраничной организации файлов. В рамках этого принципа файл рассматривается как набор последовательно расположенных страниц с данными. Таким образом, страница является минимальным блоком (квантом) данных, с которым может идти работа (физической записью). Размер страницы в может быть 8Кб.

Нетрудно понять, в чем заключается смысл постраничной организации. Этот способ представления позволяет использовать разные алгоритмы буферизации и существенно повысить эффективность работы с файлами. Однако постраничная организация в применяется в SQL Server лишь для файлов с данными, т.к. именно они имеют очень сложную структуру и на работу с ними ложится основная нагрузка. Файлы журнала транзакций, напротив, устроены достаточно просто, т.к. содержат последовательно расположенные записи о происходящих транзакциях; здесь допустима работа с отдельными записями.

Страницы файлов данных могут использоваться не только для хранения собственно данных, но и для представления различной служебной информации. Так, страницы могут быть одного из следующих типов:

- Data. Предназначены для хранения данных таблиц, кроме т.н. «тяжелых» данных (`Image`, `Text`, `nText`).
- Index. Предназначены для хранения индексов таблиц и представлений.
- Text/Image. Используются для хранения «тяжелых» данных `Image`, `Text`, `nText`, занимающих обычно более одной страницы. Для таких данных место сразу же выделяется постранично.
- Global Allocation Map (GAM). Служебный тип страницы. Содержит информацию об использовании групп страниц.
- Page Free Space (PFS). Содержит сведения о расположении свободного пространства в страницах файла. Свообразная замена «списка свободных элементов».
- Index Allocation Map (IAM). Содержат информацию о группах страниц, которые используются таблицами.

Каждая страница имеет некоторую внутреннюю архитектуру. Так, у любой страницы присутствует заголовок, содержащий служебную информацию. Структура заголовка одинакова для страниц разных типов.

Подобная страничная организация была бы не слишком удобной для хранения огромных объемов данных, а ведь современные серверные СУБД и, в частности, Microsoft SQL Server часто используются именно в таких целях. Для дополнительной оптимизации в SQL Server было введено понятие «группа страниц» (экстент). Экстент содержит 8 страниц и, следовательно, имеет размер 64Кб. При необходимости выделить новую страницу создается сразу целая пачка из 8 страниц. Наряду с различиями в типах страниц, существуют аналогичные различия и в типах экстентов. Детальное описание структуры и типов экстентов можно найти, например в.

### *16.1.3 Алгоритмы*

Алгоритмы бд основаны на максимальное использование кэша и оперативной памяти. В самом простом случае используются B-tree [14.1.7.6] (в реальности же B+ или B\*) для сохранения данных на жестком диске, тк эти деревья позволяют хранить индекс в оперативе.

Индексы стараются разместить целиком в оперативке, и для увеличения скорости чтения вместе с индексами лежит значение, для того, чтобы не лезть на жесткий диск.

Возможно сделать несколько индексов по одним и тем же данным, это позволит быстро искать по нескольким атрибутам (заголовкам таблицы).

Так же выбираются разные стратегии записи в файл. Одна из эффективных по времени – при большой нагрузке на бд аллоцировать заведомо большой файл, и писать постепенно в него, а при спаде нагрузки фрагментировать данные. Так же можно не удалять файлы, а только помечать их на удаление и удалять данные ночью при маленькой нагрузке. Однако если бд должна работать 24/7 могут возникнуть проблемы из-за того, что чистка мусора будет идти одновременно с ответами на транзакции клиента, что увеличит время ответа.

Для ускорения записи в бд можно писать данные в write-only файлы в конец, а в индексе хранить ссылку на место на диске. При этом изменение записи не удаляет старую запись, а лишь добавляет новую в конец и индекс начинает указывать на новую запись. Но если старые данные не чистить – диск забывается очень быстро, для этого можно сделать фоновую программу, которая будет мерджить файлы только по валидным данным (будет ссылка из индекса).

Однако, если в базе хранятся только чиселки (4 байта), то этот подход будет неэффективен, тк уйдет 4кб на индекс файла и 4кб на позицию в файле, получится, что индекс будет занимать места больше, чем сами данные. Тогда имеет смысл хранить в оперативе само значение. В случае больших файлов (картинок, музыки) – тот способ подойдет.

Сейчас есть зависимость 1гб оперативы на 1тб данных.

Хранение бинарного дерева в виде бинарной кучи 14.1.7.11 - кэш френдли.

## 16.2 Методы индексирования

Индекс (англ. index) — объект базы данных, создаваемый с целью повышения производительности поиска данных.

Таблицы в базе данных могут иметь большое количество строк, которые хранятся в произвольном порядке, и их поиск по заданному критерию путём последовательного просмотра таблицы строки за строкой может занимать много времени. Индекс формируется из значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы и, таким образом, позволяет искать строки, удовлетворяющие критерию поиска. Ускорение работы с использованием индексов достигается в первую очередь за счёт того, что индекс имеет структуру, оптимизированную под поиск — например, сбалансированного дерева. См 14.1.7.3

Для оптимальной производительности запросов индексы обычно создаются на тех столбцах таблицы, которые часто используются в запросах. Для одной таблицы может быть создано несколько индексов. Однако увеличение числа индексов замедляет операции добавления, обновления, удаления строк таблицы, поскольку при этом приходится обновлять сами индексы. Кроме того, индексы занимают дополнительный объем памяти, поэтому перед созданием индекса следует убедиться, что планируемый выигрыш в производительности запросов превысит дополнительную затрату ресурсов компьютера на сопровождение индекса.

Индекс – это синоним ключа. Индекс – структура данных; ключ – понятие в реляционной алгебре.

### 16.2.1 Про БД

<https://www.youtube.com/watch?v=4Tgvd6NPufs>

Для индексирования обычно создается отдельная таблица, в которой хранятся отсортированные проиндексированные данные, по которым можно получить требуемый кортеж.

Для создания индекса используют разные подходы:

#### *16.2.1.1 В дерево*

В postgres используется по умолчанию. Подробнее в 14.1.7.6.

##### **Можно:**

- Поиск по полному значению;
- Поиск по самому левому префиксу;
- Поиск по префиксу столбца;
- Поиск по диапазону значений;
- Поиск по полному совпадению одной части и диапазону в другой части;
- Запросы только по индексу.

##### **Нельзя:**

- Поиск без использования левой части ключа;
- Нельзя пропускать столбцы;
- Оптимизация после поиска в диапазоне.

#### *16.2.1.2 Hash индексы*

По своей сути это hash map. Смотри 14.1.9.

В postgres он не пишется в журнал транзакций, поэтому его лучше не использовать, тк при сбое будет рассинхрон ключей и данных. Но он отлично работает во временных таблицах, там где нету ключа.

Хеш дает коллизию, из-за чего могут быть несколько аспектов:

- Если большая коллизия, то поиск во всех элементах с одним хешем займет много времени, из-за чего потеряется преимущество быстрого расчета хеша
- Мы не можем использовать индекс без данных, нужно дергать жесткий диск.

- Нельзя использовать данные в индексе, чтобы избежать чтения строк.
- Нельзя использовать для сортировки, поскольку строки в нем не хранятся в отсортированном порядке.
- Хеш-индексы не поддерживают поиск по частичному ключу, так как хеш-коды вычисляются для всего индексируемого значения.
- Хеш-индексы поддерживают только сравнения на равенство, использующие операторы =, IN() и <=>.
- Доступ к данным в хеш-индексе очень быстр, если нет большого количества коллизий.
- Некоторые операции обслуживания индекса могут оказаться медленными, если количество коллизий велико.

#### *16.2.1.3 GiST индексы*

Хорошо, например для координат, когда нужно проверить пересечение полигонов. Основано на R-tree 14.1.7.9

В Postgres GiST позволяет создать для любого собственного типа данных индекс, основанный на R-Tree, если реализовать порядка 7 методов.

#### *16.2.1.4 GIN (инвертированный)*

Используется для полнотекстового поиска, для парса JSON, массивов, итд.

#### **Данные**

<b>id</b>	<b>title</b>	<b>genres</b>
1	Toy Story	{'Animation', 'Children', 'Comedy'}
589	Terminator 2: Judgment Day	{'Action', 'Sci-Fi'}
741	Ghost in the Shell	{'Animation', 'Sci-Fi'}
45517	Cars	{'Animation', 'Children', 'Comedy'}

#### **Индекс**

<b>key</b>	<b>ids</b>
Action	589
Animation	1, 741, 45517
Children	1, 45517
Comedy	1, 45517
Sci-Fi	589, 741

### *16.2.1.5 Функциональный индекс*

Индекс строится по функции от значения поля, например функция lower().

```
CREATE TABLE movies (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    genres TEXT[] NOT NULL
);

CREATE INDEX idx_movies_title
ON movies (LOWER(title));

SELECT * FROM movies
WHERE title = 'Alice in Wonderland';

SELECT * FROM movies
WHERE LOWER(title) = LOWER('Alice in Wonderland');
```

Но оборотная сторона медали – надо везде тащить функцию в запросах.

### *16.2.1.6 Кластерный индекс*

Этот индекс упорядочивает сами данные, а не индекс. Полезно, когда нужно добиться последовательного чтения с жесткого диска.

### *16.2.2 Особенность SQL*

Индексы полезны для многих приложений, однако на их использование накладываются ограничения. Возьмём такой запрос SQL:

```
SELECT first_name FROM people WHERE last_name = 'Франкенштейн';
```

Для выполнения такого запроса без индекса СУБД должна проверить поле last\_name в каждой строке таблицы (этот механизм известен как «полный перебор» или «полное сканирование таблицы», в плане может отображаться словом NATURAL). При использовании индекса СУБД просто проходит по В-дереву, пока не найдёт запись «Франкенштейн». Такой проход требует гораздо меньше ресурсов, чем полный перебор таблицы.

Теперь возьмём такой запрос:

```
SELECT email_address FROM customers WHERE email_address LIKE  
'%@yahoo.com';
```

Этот запрос должен нам найти всех клиентов, у которых е-мейл заканчивается на @yahoo.com, однако даже если по столбцу email\_address есть индекс, СУБД всё равно будет использовать полный перебор таблицы. Это связано с тем, что индексы строятся в предположении, что слова/символы идут слева направо. Использование символа подстановки в начале условия поиска исключает для СУБД возможность использования поиска по В-дереву. Эта проблема может быть решена созданием дополнительного индекса по выражению reverse(email\_address) и формированием запроса вида:

```
SELECT email_address FROM customers WHERE reverse(email_address) LIKE  
reverse('%@yahoo.com');
```

## 17

### **17.1 Компоненты систем управления базами данных.**

Система управления базами данных - совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.

#### *17.1.1 Основные функции:*

- управление данными во внешней памяти (на дисках);
- управление данными в оперативной памяти с использованием дискового кэша;
- журнализация изменений, резервное копирование и восстановление базы данных после сбоев;
- поддержка языков БД (язык определения данных, язык манипулирования данными).

#### *17.1.2 Состав СУБД*

Обычно современная СУБД содержит следующие компоненты:

- ядро, которое отвечает за управление данными во внешней и оперативной памяти и журнализацию;

- процессор языка базы данных, обеспечивающий оптимизацию запросов на извлечение и изменение данных и создание, как правило, машинно-независимого исполняемого внутреннего кода;
- подсистему поддержки времени исполнения, которая интерпретирует программы манипуляции данными, создающие пользовательский интерфейс с СУБД;
- сервисные программы (внешние утилиты), обеспечивающие ряд дополнительных возможностей по обслуживанию информационной системы.

В основу архитектуры БД легла архитектура ANSI — SPARC [Ошибка! Источник ссылки не найден.] в основе которой лежит три уровня системы:

- Внешний (пользовательский)
- Концептуальный (промежуточный)
- Внутренний (физический).

В БД ключевые компоненты это:

- Таблицы
- Индексы
- Триггеры
- Функции
- Лог транзакций
- Настройки (например в которых есть видимость параллельных транзакций)

## 17.2 Целостность данных.

См 18.3.

## 17.3 Транзакции.

Транзакция (англ. transaction, от лат. transactio — соглашение, договор) — минимальная логически осмысленная операция, которая имеет смысл и может быть совершена только полностью.

Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной.

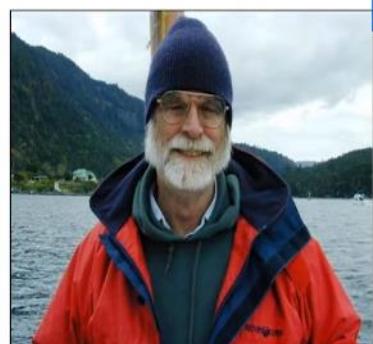
Транзакции нужны для обеспечения предсказуемости системы в случае многопользовательского использования и в особенностях работы оборудования и возникновения внештатных ситуаций.

В информатике акроним ACID описывает требования к транзакционной системе (например, к СУБД), обеспечивающие наиболее надёжную и предсказуемую её работу.

## ACID

ACID описывает требования к транзакционной системе, обеспечивающие наиболее надёжную и предсказуемую её работу. Требования ACID были в основном сформулированы в конце 70-х годов Джимом Греем.

- Atomicity – Атомарность
- Consistency – Согласованность
- Isolation – Изолированность
- Durability – Долговечность



<https://www.youtube.com/watch?v=XkS3937Xn8M&list=PLrCZzMib1e9oOFQbuOgjKYbRUoA8zGKnj&index=4>

Атомарность – за единицу работы СУБД берется транзакция. Либо она вся отработала, либо она совсем не отработала. Все или ничего.

Согласованность – каждая успешная транзакция по определению фиксирует только допустимые результаты.

Изолированность – во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат.

Долговечность – независимо от проблем на нижних уровнях (выкл питания или сбой) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранными после возвращения в работу.

# Пример транзакции

```
BEGIN;
```

```
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
```

```
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
```

```
COMMIT;
```

## 17.3.1 Атомарность (Журнал транзакций)

Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной.

В простейшем случае журнализация изменений заключается в последовательной записи всех изменений, выполняемых в базе данных.

Записывается следующая информация:

- порядковый номер, тип и время изменения;
- идентификатор транзакции;
- объект, подвергшийся изменению (номер хранимого файла и номер блока данных в нём, номер строки внутри блока);
- предыдущее состояние объекта и новое состояние объекта.

Журнал содержит отметки начала и завершения транзакции, и отметки принятия контрольной точки.

Каждая операция не происходит с данными напрямую. Идет принцип двойной записи: сначала в журнал, потом в данные.

Это делается, тк:

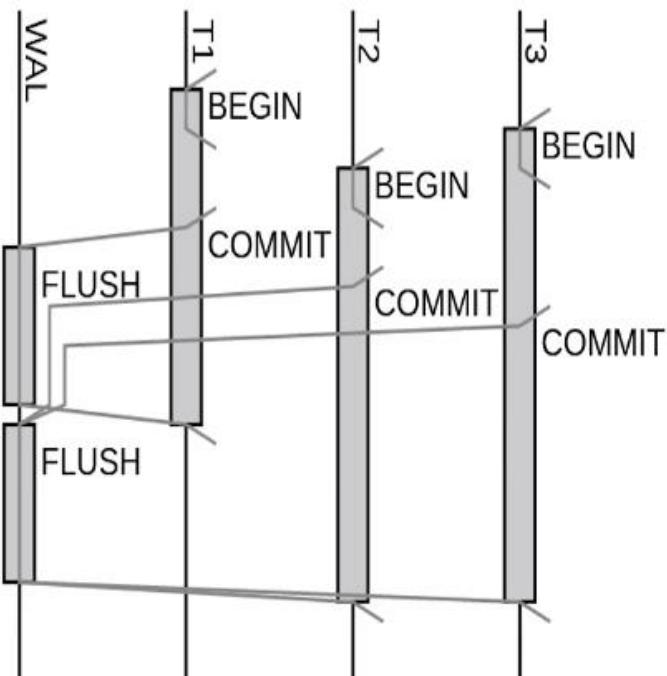
- запись данных на диск не имеет атомарную функцию записи
- запись на диск сильно медленнее записи в оперативу

## Общий алгоритм:

- изменения пишутся в журнал транзакций (состояние до и после) и изменяются страницы БД в памяти;
- в журнал транзакций сбрасывается маркер успешного завершения транзакции;
- журнал транзакций сбрасывается на диск;
- данные страниц БД пишутся на диск;
- данные страниц БД сбрасываются на диск.



Для быстродействия диска транзакции пишутся на диск группой, как на пикче ниже



Журнал транзакций позволяет восстановить данные «Point in time recovery».

Важная тонкость для postgres: индекс hashmap не пишется в журнал транзакций (сейчас вроде исправили).

### *17.3.2 Согласованность*

Каждая успешная транзакция по определению фиксирует только допустимые результаты.

Согласованность является более широким понятием. Например, в банковской системе может существовать требование равенства суммы, списываемой с одного счёта, сумме, зачисляемой на другой. Это бизнес-правило и оно не может быть гарантировано только проверками целостности, его должны соблюсти программисты при написании кода транзакций. Если какая-либо транзакция произведёт списание, но не произведёт зачисления, то система останется в некорректном состоянии и свойство согласованности будет нарушено.

### *17.3.3 Изолированность*

#### *17.3.3.1 MVCC*

Она обеспечивается MVCC – MultiVersion Concurrency Control / Управление параллельным доступом посредством многоверсионности

#### **MVCC**

MultiVersion Concurrency Control

- Разные пользователи могут одновременно работать с одними и теми же данными;
- Каждый пользователь видит свой изолированный срез данных;
- Изменения, вносимые пользователем, никому не видны до завершения транзакции.

## MVCC 1: TABLE

xmin	xmax	id	name	notes
100	0	1	Alice	Great at programming
101	0	2	Bob	Always talk to alice
102	0	3	Eve	Listens to everyone's conversations

TXID: 103

## MVCC 2: UPDATE

xmin	xmax	id	name	notes
100	0	1	Alice	Great at programming
- update	101	0	Bob	Always talk to alice
102	0	3	Eve	Listens to everyone's conversations

TXID: 103

## MVCC 3: UPDATE IN PROGRESS

xmin	xmax	id	name	notes
100	0	1	Alice	Great at programming
- update	101	103	Bob	Always talk to alice
102	0	3	Eve	Listens to everyone's conversations

TXID: 103

## MVCC 4: UPDATE IN PROGRESS

	xmin	xmax	id	name	notes	TXID: 103
- update	100	0	1	Alice	Great at programming	
	101	103	2	Bob	Always talk to alice	
	102	0	3	Eve	Listens to everyone's conversations	
+ update	103	0	2	Bob	Working very hard	

## MVCC 5: UPDATED

	xmin	xmax	id	name	notes	TXID: 104
+ update	100	0	1	Alice	Great at programming	
	101	103	2	Bob	Always talk to alice	
	102	0	3	Eve	Listens to everyone's conversations	
	103	0	2	Bob	Working very hard	

## MVCC 6: INSERT

xmin	xmax	id	name	notes	TXID: 104
100	0	1	Alice	Great at programming	
101	<b>103</b>	2	Bob	<i>Always talk to alice</i>	
102	0	3	Eve	Listens to everyone's conversations	
<b>103</b>	0	2	Bob	<b>Working very hard</b>	
+ insert		104	Dave	<b>Very promising new-hire</b>	

## MVCC 7: INSERTED

xmin	xmax	id	name	notes	TXID: 105
100	0	1	Alice	Great at programming	
101	<b>103</b>	2	Bob	<i>Always talk to alice</i>	
102	0	3	Eve	Listens to everyone's conversations	
<b>103</b>	0	2	Bob	<b>Working very hard</b>	
	0	4	Dave	<b>Very promising new-hire</b>	

## MVCC 8: DELETE

- delete →

xmin	xmax	id	name	notes	TXID: 105
100	0	1	Alice	Great at programming	
101	<b>103</b>	2	Bob	<i>Always talk to alice</i>	
102	105	3	Eve	<i>Listens to everyone's conversations</i>	
103	0	2	Bob	<b>Working very hard</b>	
104	0	4	Dave	<b>Very promising new-hire</b>	

## MVCC 9: DELETED

xmin	xmax	id	name	notes	TXID: 106
100	0	1	Alice	Great at programming	
101	<b>103</b>	2	Bob	<i>Always talk to alice</i>	
102	105	3	Eve	<i>Listens to everyone's conversations</i>	
103	0	2	Bob	<b>Working very hard</b>	
104	0	4	Dave	<b>Very promising new-hire</b>	

### 17.3.3.2 Уровни изолированности транзакций

## Уровни изолированности транзакций

- Read uncommitted - чтение незафиксированных данных
- Read committed (по-умолчанию) - чтение фиксированных данных
- Repeatable read - повторяемость чтения
- Serializable - упорядочиваемость

Уровень изоляции	Потерянное обновление	«Грязное» чтение	Неповторяющееся чтение	Фантомное чтение	Аномалии сериализации
Read uncommitted	не возможно	допускается	возможно	возможно	возможно
Read committed	не возможно	не возможно	возможно	возможно	возможно
Repeatable read	не возможно	не возможно	не возможно	допускается	возможно
Serializable	не возможно	не возможно	не возможно	не возможно	не возможно

## Потерянное обновление (Lost Update)

Потерянное обновление происходит в случае перезатирания изменений другой транзакцией до завершения транзакции, сделавшей изменения.

Транзакция 1	Транзакция 2
UPDATE tbl1 SET f2=f2+20 WHERE f1=1;	
	UPDATE tbl1 SET f2=f2+25 WHERE f1=1;

## «Грязное» чтение (Dirty Read)

Чтение данных, добавленных или изменённых еще не завершённой транзакцией.

Транзакция 1	Транзакция 2
<code>SELECT f2 FROM tbl1 WHERE f1=1; f2 ----- 100 (1 row)</code>	
<code>UPDATE tbl1 SET f2=200 WHERE f1=1;</code>	
	<code>SELECT f2 FROM tbl1 WHERE f1=1; f2 ----- 200 (1 row)</code>
<code>ROLLBACK;</code>	

## Неповторяющееся чтение (Non-Repeatable Read)

При повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными.

Транзакция 1	Транзакция 2
	<code>SELECT f2 FROM tbl1 WHERE f1=1;</code>
<code>UPDATE tbl1 SET f2=f2+1 WHERE f1=1;</code>	
<code>COMMIT;</code>	
	<code>SELECT f2 FROM tbl1 WHERE f1=1;</code>

# Чтение "фантомов" (Phantom Reads)

Ситуация, когда при повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк.

От неповторяющегося чтения оно отличается тем, что результат повторного обращения к данным изменился не из-за изменения/удаления самих этих данных, а из-за появления новых (phantomных) данных.

Транзакция 1	Транзакция 2
	<code>SELECT SUM(f2) FROM tbl1;</code>
<code>INSERT INTO tbl1 (f1,f2) VALUES (15,20);</code>	
<code>COMMIT;</code>	
	<code>SELECT SUM(f2) FROM tbl1;</code>

## Аномалии сериализации

Ситуация, когда параллельное выполнение транзакций приводит к результату, невозможному при последовательном выполнении тех же транзакций.

Транзакция 1	Транзакция 2
<code>SELECT SUM(value) FROM mytab WHERE class = 1;</code> ----- 30 (1 row)	<code>SELECT SUM(value) FROM mytab WHERE class = 2;</code> ----- 300 (1 row)
<code>INSERT INTO mytab (value, class) VALUES (30, 2)</code>	<code>INSERT INTO mytab (value, class) VALUES (300, 1)</code>
<code>COMMIT;</code>	<code>COMMIT;</code>

### 17.3.4 Долговечность

Независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу.

## 18.1 Архитектура систем баз данных

Архитектура ANSI-SPARC (также трёхуровневая архитектура) определяет принцип, согласно которому рекомендуется строить системы управления базами данных (СУБД).

Основное внимание в этом подходе сконцентрировано на необходимости воплощения независимости каждого уровня для изоляции программ от особенностей представления данных на более низком уровне.

Три уровня системы:

- Внешний (пользовательский)
- Концептуальный (промежуточный)
- Внутренний (физический).

В основе архитектуры ANSI-SPARC лежит **концептуальный** уровень. В современных СУБД он может быть реализован при помощи представления. Концептуальный уровень описывает данные и их взаимосвязи с наиболее общей точки зрения, — концепции архитекторов базы, используя реляционную или другую модель. Туда входят:

- все сущности, включаемые в базу, их атрибуты и связи.
- Накладываемые на данные ограничения (типы данных)
- Семантическая инфа о данных
- Информация об уровне доступа и поддержки целостности данных

Концептуальный уровень поддерживает каждое внешнее представление, в том смысле, что любые доступные пользователю данные должны содержаться (или могут быть вычислены) на этом уровне. Однако этот уровень не содержит никаких сведений о методах хранения данных.

**Внутренний** уровень позволяет скрыть подробности физического хранения данных (носители, файлы, таблицы, триггеры ...) от концептуального уровня. Отделение внутреннего уровня от концептуального обеспечивает так называемую физическую независимость данных. На этом уровне хранится такая инфа, как:

- Распределение дискового пространства для хранения данных и индексов
- Описание подробностей хранения данных
- Распределение данных по дискам/кластерам

На **внешнем** уровне описываются различные подмножества элементов концептуального уровня для представлений данных различным пользовательским программам. Каждый пользователь получает в своё распоряжение часть представлений о данных, но полная концепция скрыта. Отделение внешнего уровня от концептуального обеспечивает логическую независимость данных.

## 18.2 Независимость данных

Независимость данных является типом прозрачности данных , что имеет значение для централизованной СУБД . Это относится к иммунитету пользовательских приложений на изменения , сделанные в определении и организации данных. Прикладные программы не должны, в идеале, быть подвержены деталям представления и хранения данных. СУБД обеспечивает абстрактное представление данных , который скрывает такие детали.

Есть два типа независимости данных

- логической
- физической.

Логическая независимость данных означает, что общая логическая структура данных может быть изменена без изменения прикладных программ. Логическая независимость допускает возможность применения одной концептуальной модели различными пользователями.

Физическая независимость данных означает, что физическое расположение и организация данных могут изменяться, не вызывая при этом изменений ни общей логической структуры данных, ни прикладных программ. Физическая независимость дает возможность в целях эффективности использования БД модифицировать физическую организацию данных и пути доступа. *Например, необходимо добавить или удалить некоторую связь между записями без изменения программы.*

## 18.3 Целостность данных

В теории баз данных под целостностью понимают свойство соответствия структуры и содержания базы данных предметной области. В реляционной модели данных определяются два основных требования, при которых обеспечивается целостность данных: **целостность сущностей и целостность ссылок.**

Каждый объект или наблюдение представляется в реляционной базе как группа взаимосвязанных элементов данных — кортеж некоторого отношения. Требование целостности **сущностей** заключается в том, чтобы каждый кортеж любого отношения отличался от другого кортежа этого отношения (т.е. любое отношение должно обладать первичным ключом).

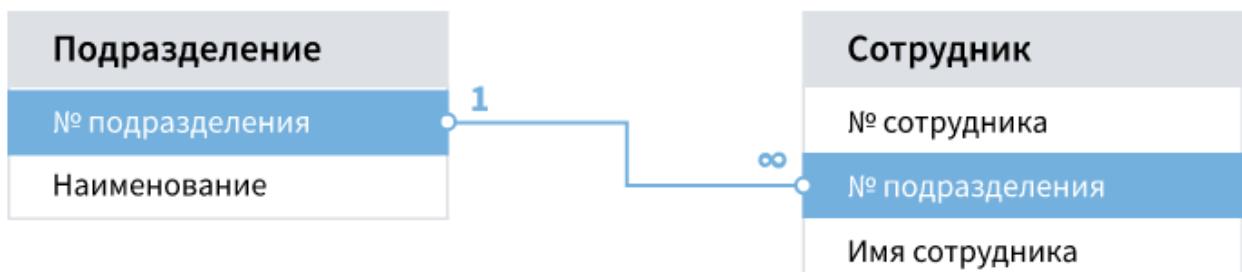
Очевидно, что если данное требование будет нарушено (т.е. кортежи в рамках одного отношения не уникальны), то в базе будет храниться противоречивая информация об одном и том же объекте.

Поддержание целостности сущностей обеспечивается средствами системы управления базой данных (СУБД) с помощью двух ограничений:

- при добавлении записей в таблицу проверяется уникальность их первичных ключей;
- запрещено изменять значения атрибутов, входящих в первичный ключ.

Требование **ссылочной** целостности состоит в том, что для каждого значения внешнего ключа, появляющегося в дочернем отношении, в родительском должен найтись кортеж с таким же значением первичного ключа.

Например, даны отношения «ПОДРАЗДЕЛЕНИЕ» («N ПОДРАЗДЕЛЕНИЯ», «НАИМЕНОВАНИЕ ПОДРАЗДЕЛЕНИЯ») и «СОТРУДНИК» («N СОТРУДНИКА», «N ПОДРАЗДЕЛЕНИЯ», «ИМЯ СОТРУДНИКА»), в которых хранятся сведения о работниках организации и подразделениях, где они работают. Отношение «ПОДРАЗДЕЛЕНИЕ» в данной паре является родительским, поэтому его первичный ключ «N ПОДРАЗДЕЛЕНИЯ» присутствует в дочернем отношении «СОТРУДНИК».



Требование ссылочной целостности означает в данном случае, что в таблице «СОТРУДНИК» не может присутствовать кортеж со значением атрибута «N ПОДРАЗДЕЛЕНИЯ», которое не встречается в таблице «ПОДРАЗДЕЛЕНИЕ». Если такое значение в отношении «ПОДРАЗДЕЛЕНИЕ» отсутствует, значение внешнего ключа в отношении «СОТРУДНИК» считается неопределенным.

Как правило, поддержание ссылочной целостности также является функцией СУБД. Например, она может запретить пользователю добавлять запись, содержащую внешний ключ с несуществующим (неопределенным) значением.

Целостность данных является важнейшим фактором их качества. Нарушение целостности не позволит производить их корректную интеграцию в хранилище данных и осуществлять их дальнейший анализ. Поэтому при реализации аналитических проектов необходимо использовать механизмы автоматического контроля и поддержания целостности данных.

## **18.4 Не избыточное хранение данных**

Для достижения не избыточности данных в реляционных БД используются нормальные формы, позволяющие избегать повторений данных [15.3]. Но чем выше нормальная форма, тем медленнее будут выполняться сложные запросы. Возможно, идеальный вариант для хранения неповторяющихся данных – это сделать таблицы только с ключом для поиска и одним значением, но join множества таких таблиц может обойтись очень дорого.

Поэтому важен баланс между нормальной формой и скоростью реакции системы на внешнее событие.

**19**

## **19.1 Язык SQL. Средства описания данных, определения ограничений целостности**

**20**

## **20.1 Язык SQL. Средства манипулирования данными.**

**21**

## **21.1 Язык XML.**

Xml – поток данных (байт), который предварительно размечен.

### *21.1.1 Xml vs Json*

Прелести XML:

- инструмент валидации схемы;
- ссылки на др элементы

Прелести JSON:

- Компактнее

## **21.2 Структурная модель документа (DTD).**

## **21.3 Адресация содержания XML-документов согласно спецификации Xlink/Xpointer/XPath.**