1. Теорема Поста о полноте систем функций в алгебре логики.

Определение. Множество функций алгебры логики *A* называется *полной системой* (в P_2), если любую функцию алгебры логики можно выразить формулой над *A*.

Определение. Пусть A ⊆ P_2. Тогда замыканием A называется множество всех функций алгебры логики, которые можно выразить формулами над A.

Обозначение: [A].

Определение: замкнутый класс – A = [A].

Стандартные замкнутые классы: $T_0\{f(0,\dots,0)=0\}$; $T_1\{f(1,\dots,1)=1\}$; $L-$ линейные $\{f(x_1,\dots,x_n)=a_0\oplus a_1x_1\oplus\dots\oplus a_nx_n\}$; $S-$ самодвойственные $\{f(x_1,\dots,x_n)=f^*(x_1,\dots,x_n)=\bar{f}(\overline{x_1},\dots,\overline{x_n})\}$; $M-$ монотонные $\{\alpha\le\beta\Rightarrow f(\alpha)\le f(\beta)\}$

Теорема Поста. Система функций алгебры логики A = {f_1, f_2, ...} является полной в P_2 тогда и только тогда, когда она не содержится целиком ни в одном из следующих классов: T_0, T_1, S, L, M.

Доказательство.

Необходимость. Пусть A — полная система, N — любой из классов T_0, T_1, S, L, M и пусть A ⊆ N. Тогда [A] ⊆ [N] = N ≠ P_2 и [A] ≠P_2. Полученное противоречие завершает обоснование необходимости.

Достаточность. Пусть $A\not\subset T_0, A\not\subset T_1, A\not\subset M, A\not\subset L, A\not\subset S$. Тогда в A существуют функции $f_0\notin T_0, f_1\notin T_1, f_M\notin M, f_L\notin L, f_S\notin S$. Достаточно показать, что $[A]\supseteq[f_0,f_1,f_M,f_L,f_S]=P_2$. Разобьём доказательство на три части: получение отрицания, констант и конъюнкции.

a) Получение $\bar{x}$. Рассмотрим функцию $f_0(x_1,\dots,x_n)\notin T_0$ и введём функцию $\varphi_0(x)=f_0(x,x,\dots,x)$. Так как функция f_0 не сохраняет нуль, φ_0 (0) = f (0, 0, ..., 0) = 1. Возможны два случая: либо $\varphi_0(x)=\bar{x}$, либо φ_0 (x) ≡ 1. Рассмотрим функцию $f_1(x_1,\dots,x_n)\notin T_1$ и аналогичным образом введём функцию $\varphi_1(x)=f_1(x,x,\dots,x)$. Так как функция f_1 не сохраняет единицу, φ_1 (1) = f (1, 1, ..., 1) = 0. Возможны также два случая: либо $\varphi_1(x)=\bar{x}$, либо φ_1 (x) ≡ 0. Если хотя бы в одном случае получил о с ь искомое отрицание, пункт завершен. Если же в обоих случаях получились константы, то согласно лемме о немонотонной функции, подставляя в функцию $f_M\notin M$ вместо всех переменных константы и тождественные функции, можно получить отрицание. Отрицание получено.

b) Получение констант 0 и 1. Имеем $f_S\notin S$. Согласно лемме о несамодвойственной функции, подставляя вместо всех переменных функции f_S отрицание (которое получено в пункте а) и тождественную функцию, можно получить константы: $[f_S,\bar{x}]\ni[0,1]$. Константы получены.

c) Получение конъюнкции x · y. Имеем функцию $f_L\notin L$. Согласно лемме о нелинейной функции, подставляя в функцию f_L вместо всех переменных константы и отрицания (которые были получены на предыдущих шагах доказательства), можно получить либо конъюнкцию, либо отрицание конъюнкции. Однако на первом этапе отрицание уже получено, следовательно, всегда можно получить конъюнкцию: $[f_L,0,1,\bar{x}]\ni[xy,\overline{xy}]$. Конъюнкция получена.

В результате получено, что $[f_0,f_1,f_M,f_L,f_S]\supseteq[\bar{x},xy]=P_2$.

2. Графы, деревья, планарные графы; их свойства. Оценка числа деревьев.

Определение 1. *Графом* называется произвольное множество элементов V и произвольное семейство E пар из V. Обозначение: $G = (V, E)$.

Определение 2. Если элементы из E рассматривать как неупорядоченные пары, то граф называется *неориентированным*, а пары называются *рёбрами*. Если же элементы из E рассматривать как упорядоченные, то граф *ориентированный*, а пары — *дуги*.

Определение 3. Пара вида (a, a) называется *петлёй*, если пара (a, b) встречается в семействе E несколько раз, то она называется *кратным ребром* (*кратной дугой*).

Определение 4. В дальнейшем условимся граф без петель и кратных рёбер называть *неориентированным графом* (или просто *графом*), граф без петель — *мультиграфом*, а мультиграф, в котором разрешены петли — *псевдографом*.

Определение 5. Две вершины графа называются *смежными*, если они соединены ребром.

Определение 6. Говорят, что вершина и ребро *инцидентны*, если ребро содержит вершину.

Определение 7. *Степенью вершины* ($\deg v$) называется количество рёбер, инцидентных данной вершине. Для псевдографа полагают учитывать петлю дважды. 25

Утверждение 1. В любом графе (псевдографе) справедливо следующее соотношение

$$\sum_{i=1}^{p} \deg v_i = 2q, \text{где } p - \text{число вершин}, q - \text{число ребер}$$

Доказательство. Когда мы считаем степень одной вершины, мы считаем все рёбра, выходящие из неё. Вычисляя сумму всех степеней, мы получаем, что каждое ребро считается дважды, так как оно инцидентно двум вершинам (петли по определению степени также посчитаются дважды). Поэтому общая сумма будет равна удвоенному числу рёбер. Утверждение доказано.

Определение 1. *Деревом* называется связный граф без циклов.

Определение 2. Подграф $G_1 = (V_1, E_1)$ графа $G = (V, E)$, называется *остовным деревом* в графе $G = (V, E)$, если $G_1 = (V_1, E_1)$ — дерево и $V_1 = V$.

Лемма 1. Если граф $G = (V, E)$ связный и ребро (a, b) содержится в некотором цикле в графе G, то при выбрасывании из графа G ребра (a, b) снова получится связный граф.

Доказательство. Это утверждение следует из того, что при выбрасывании из графа G ребра (a, b) вершины a и b всё равно остаются в одной связной компоненте, поскольку из a в b можно пройти по оставшейся части цикла. Лемма доказана.

Теорема 1. Любой связный граф содержит хотя бы одно остовное дерево.

Доказательство. Если в G нет циклов, то G является искомым остовным деревом. Если в G есть циклы, то удалим из G какое-нибудь ребро, входящее в цикл. Получится некоторый подграф G_1. По лемме 1 G_1 — связный граф. Если в G_1 нет циклов, то G_1 и есть искомое остовное дерево, иначе продолжим этот процесс. Процесс должен завершиться, так как E — конечное множество. Теорема доказана.

Лемма 2. Если к связному графу добавить новое ребро на тех же вершинах, то появится цикл.

Доказательство. Рассмотрим произвольный связный граф $G = (V, E)$. Пусть $u \in V, v \in V, (u, v) \notin E$. Так как G — связный граф, то в нём есть путь из v в u. Тогда в G есть и простая цепь C из v в u. Поэтому в полученном графе есть цикл C, (u, v), v. Лемма доказана.

Лемма 3. Пусть в графе $G = (V, E)$ p вершин и q рёбер. Тогда в G не менее $p - q$ связных компонент. Если при этом в G нет циклов, то G состоит ровно из $p - q$ связных компонент.

Доказательство. Пусть к некоторому графу H, содержащему вершины u и v, добавляется ребро (u, v). Тогда если u и v лежат в разных связных компонентах графа H, то число связных компонент уменьшится на 1. Если u, v лежат в одной связной компоненте графа H, то число связных компонент не изменится. В любом случае, число связных компонент уменьшается не более чем на 1. Значит, при добавлении q рёбер число связных компонент уменьшается не более чем на q. Так как граф G получается из графа $G_1 = (V, \emptyset)$ добавлением q рёбер, то в G не менее $p - q$ связных компонент.

Пусть теперь в G нет циклов, и пусть в процессе получения G из G_1 добавляется ребро (u, v). Если бы u, v лежали уже в одной связной компоненте, то в G, согласно лемме 2, возникал бы цикл. Следовательно, u, v лежат в разных связных компонентах и при добавлении ребра (u, v) число связных компонент уменьшается ровно на 1. Тогда G состоит ровно из $p - q$ связных компонент. Лемма доказана.

Определение 1. Любое дерево, в котором выделена одна вершина, называемая *корнем*, называется *корневым деревом*.

Определение 2.

1) Граф, состоящий из одной вершины, которая выделена, называется *корневым деревом*.

2) Пусть имеются корневые деревья $D_1, D_2, ..., D_m$ с корнями $v_1, v_2, ..., v_m$, $D_i = (V_i, E_i)$, $V_i \cap V_j = \emptyset$ $(i \neq j)$. Тогда граф $D = (V, E)$, полученный следующим образом:

$V = V_1 \cup V_2 \cup ... \cup V_m \cup \{v\} (v \notin V_i, \forall i), E = E_1 \cup E_2 \cup ... \cup E_m \cup \{(v, v_1), (v, v_2), \ldots, (v, v_m)\}$

и в котором выделена вершина v, называется *корневым деревом*.

3) Только те объекты являются *корневыми деревьями*, которые можно построить согласно пунктам 1) и 2).

Определение 3. *Упорядоченным корневым деревом* называется корневое дерево, в котором
1) задан порядок поддеревьев и
2) каждое поддерево D_i является упорядоченным поддеревом.

Дерево с одной вершиной также является упорядоченным поддеревом.

Теорема 3. Число упорядоченных корневых деревьев с q рёбрами не превосходит 4^q.

Доказательство. Рассмотрим алгоритм обхода упорядоченного дерева, называемого «поиском в глубину». Этот обход описывается рекурсивно следующим образом:
1) Начать с корня. Пока есть поддеревья выполнять:
2) перейти в корень очередного поддерева, обойти это поддерево «в глубину».
3) Вернуться в корень исходного поддерева.

В результате обход «в глубину» проходит по каждому ребру дерева ровно 2 раза: один раз при переходе в очередное поддерево, второй раз при возвращении из этого поддерева. В соответствии с обходом «в глубину» будем строить последовательность из нулей и единиц, записывая на каждом шаге нуль или единицу, причём нуль будем записывать, если происходит переход в очередное поддерево, а единицу, если мы возвращаемся из поддерева. Получим последовательность из 0 и 1 длины $2q$, которую назовём кодом дерева. По этому коду однозначно восстанавливается дерево, поскольку каждый очередной разряд однозначно указывает, начинать ли строить новое очередное поддерево или возвращаться на ярус ближе к корню. Таким образом, упорядоченных корневых деревьев с q рёбрами не больше, чем последовательностей из 0 и 1 длины $2q$, а их число равно $2^{2q} = 4^q$. Теорема доказана.

Изоморфизм корневых деревьев определяется так же, как и изоморфизм графов, но с дополнительным требованием: корень должен отображаться в корень. Для упорядоченных корневых деревьев также требуется сохранение порядка поддеревьев.

Следствие. Число неизоморфных корневых деревьев с q рёбрами и число неизоморфных деревьев с q рёбрами не превосходит 4^q.

Доказательство. Выделяя в неизоморфных деревьях по одной вершине, мы получим неизоморфные корневые деревья. Упорядочивая поддеревья в неизоморфных корневых деревьях, мы получим различные упорядоченные корневые деревья. Поэтому число неизоморфных деревьев с q рёбрами не превосходит числа неизоморфных корневых деревьев с q рёбрами, которое, в свою очередь, не превосходит числа различных упорядоченных корневых деревьев с q рёбрами. Отсюда и из теоремы следует утверждение следствия. Следствие доказано.

Определение 1. Граф называется *планарным*, если существует его геометрическая реализация на плоскости.

Определение 2. Если имеется планарная реализация графа и мы «разрежем» плоскость по всем линиям этой планарной реализации, то плоскость распадётся на части, которые называются *гранями* этой планарной реализации (одна из граней бесконечна, она называется *внешней гранью*).

Формула Эйлера. Для любой планарной реализации связного планарного графа $G = (V, E)$ с p вершинами, q рёбрами и r гранями выполняется равенство: $p - q + r = 2$.

Доказательство. Докажем теорему при фиксированном p индукцией по q. Так как G — связный граф, то $q \geq p - 1$.

a) Базис индукции: $q=p-1$.Так как G—связный и $q=p-1$,то согласно пункту 3 теоремы 2 G — дерево, то есть, в G нет циклов. Тогда $r = 1$. Отсюда $p - q + r = = p - (p - 1) + 1 = 2$.

b) Пусть для q: $p - 1 \leq q < q_0$ теорема справедлива. Докажем, что для $q = q_0$ она также справедлива. Пусть G — связный граф с p вершинами и q_0 рёбрами и пусть в его планарной реализации r граней. Так как $q_0 > p - 1$, то G — не дерево. Следовательно, в G есть цикл. Пусть ребро e входит в цикл. Тогда к нему с двух сторон примыкают разные грани. Удалим ребро e из G. Тогда две грани сольются в одну, а полученный граф G_1 останется связным. При этом получится планарная реализация графа G_1 с p вершинами и $q_0 - 1$ рёбрами и $r - 1$ гранями. Так как $q_0 - 1 < q_0$, то, по предположению индукции, для G_1 справедлива формула Эйлера, то есть $p - (q_0 - 1) + (r - 1) = 2$, откуда $p - q_0 + r = 2$. Что и требовалось доказать.

3. Логика 1-го порядка. Выполнимость и общезначимость. Общая схема метода резолюций.

Терм – переменная, константа, составной терм (функция от термов).
Формула – атомарная $P^{(m)}(t_1, \dots, t_n)$ $P^{(m)} \in Pred, t_1, \dots, t_n \in Term$, составная.
Интерпретация сингнатуры <Const, Func, Pred> – алгебраическая система $I =< D_I, \overline{Const}, \overline{Func}, \overline{Pred} >$ где

- D_I – непустое множество, которое называется областью интерпретации
- $\overline{Const}$ – оценка констант, сопоставляет каждой константе предмет из области интерпретации
- $\overline{Func}$ – оценка функциональных символов – сопоставляет n-местной функции f функцию из области интерпретации
- $\overline{Pred}$ – оценка предикатных символов – сопоставляет каждому предикатному символу отношение на области интерпретации

Отношение выполнимости: |=
I – интерпретация. $I \mid = \phi(x_1, \dots, x_n)[d_1, \dots, d_n]$, т.е. формула выполнима в интерпретации I на наборе $d_1, \dots, d_n$ если

1) Если $\phi(x_1, \dots, x_n) = P(t_1, \dots, t_m)$, то выполнимость равносильна $\bar{P}(t_1[d_1, \dots, d_n], \dots, t_m[d_1, \dots, d_n]) = true$
2) $\phi \& \psi$ <=> обе выполнимы на наборе
3) $\phi \vee \psi$ – хотя бы одна выполнима на наборе
4) $\phi \to \psi$ – либо выполнима на наборе вторая, либо невыполнима на наборе первая.
5) $\neg\phi$ – не выполнима на наборе

Формула выполнима в интерпретации, если существует хотя бы один набор элементов интерпретации, на котором формула выполнима.
Формула истинна в интерпретации, если она выполнима на любом наборе элементов интерпретации.
Формула выполнима, если существует интерпретация, в которой она выполнима.
Формула общезначима, если она истинна в любой интерпретации.
Формула невыполнима, если она не является выполнимой.
Общая схема метода резолюций:
Задача – проверить формулу на общезначимость.

1) Свести проблему общезначимости к проблеме противоречивости: если формула общезначима, то ее отрицание невыполнимо
2) Построить предваренную нормальную форму (ПНФ) $Q_1x_1Q_2x_2 \dots Q_nx_n (D_1 \& D_2 \& \dots \& D_N)$
3) Построить сколемовскую стандартную форму (ССФ) $\forall x_{i1} \forall x_{i2} \dots \forall x_{ik} (D_1 \& D_2 \& \dots \& D_N)$
4) Построить систему дизъюнктов $S_\varphi = \{D_1, D_2, \dots, D_N\}$
5) Резолютивно вывести из системы пустой дизъюнкт, тем самым доказав противоречивость. Правило резолюции:

$$Res: \frac{D_1 = D_1' \vee L, D_2 = D_2' \vee L}{D_0 = D_1' \vee D_2'}$$

4. Логическое программирование. Декларативная семантика и операционная семантика; соотношение между ними. Стандартная стратегия выполнения логических программ.

Синтаксис логических программ

Пусть $\sigma = \langle$ Const,Func,Pred$\rangle$ — некоторая сигнатура, в которой определяются термы и атомы.

«заголовок» ::= «атом»

«тело» ::= «атом» | «тело», «атом»

«правило» ::= «заголовок» ← «тело»;

«факт» ::= «заголовок»;

«утверждение» ::= «правило» | «факт»

«программа» ::= «пусто» | «утверждение» «программа»

«запрос» ::= □ | ? «тело»

Декларативная семантика	Операционная семантика
Правило $A_0 \leftarrow A_1,A_2,...,A_n$;	
Если выполнены условия $A_1,A_2,...,A_n$, то справедливо и утверждение A_0.	Чтобы решить задачу A_0 , достаточно решить задачи $A_1,A_2,...,A_n$.
Факт A_0;	
Утверждение A_0 считается верным.	Задача A_0 объявляется решенной.
Запрос $?C_1,C_2,...,C_m$	
При каких значениях целевых переменных будут верны все отношения $C_1, C_2, \ldots, C_m$?	Решить список задач $C_1,C_2,...,C_m$.

С точки зрения декларативной семантики,

- программные утверждения D и запросы G — это логические формулы,
- программа P — это множество формул (база знаний),
- а правильный ответ на запрос — это такие значения переменных (подстановка), при которой запрос оказывается логическим следствием базы знаний.

Под операционной семантикой понимают правила построения вычислений программы.

Операционная семантика описывает, КАК достигается результат работы программы.

Результат работы логической программы — это правильный ответ на запрос к программе. Значит, операционная семантика должна описывать метод вычисления правильных ответов.

Запрос к логической программе порождает задачу о логическом следствии. Значит, вычисление ответа на запрос должно приводить к решению этой задачи.

Таким методом вычисления может быть разновидность метода резолюций, учитывающая особенности устройства программных утверждений

Пусть $G = ? C_1,...,C_i,...,C_m$ — целевое утверждение, в котором выделена подцель C_i ,

$D' = A'_0 \leftarrow A'_1,A'_2,...,A'_n$ —вариант некоторого программного утверждения, в котором $Var_G \cap Var_{D'} = \emptyset$,

$\theta \in НОУ(C_i , A'_0)$ — наиболее общий унификатор подцели C_i и заголовка программного утверждения A_0. Тогда запрос $G' = ?(C_1,...,C_{i-1},A'_1,A'_2,...,A'_n,C_{i+1},...,C_m)\theta$ называется SLD-резольвентой программного утверждения D' и запроса G с выделенной подцелью C_i и унификатором θ.

Пусть $G_0 = ? C_1,C_2,...,C_m$ — целевое утверждение, $P=\{D_1,D_2,...,D_N\}$—хорновская логическая программа.

Тогда (частичным) SLD-резолютивным вычислением , порожденным запросом G_0 к логической программе P называется последовательность троек (конечная или бесконечная)
$(D_{j1},\theta_1,G_1),(D_{j2},\theta_2,G_2),...,(D_{jn},\theta_n,G_n),...,$
в которой для любого i, $i \geq 1$,

> $D_{ji} \in P$, $\theta_i \in Subst$, G_i — целевое утверждение (запрос);

> запрос G_i является SLD-резольвентой программного утверждения D_{ji} и запроса G_{i-1} с унификатором θ_i.

Частичное SLD-резолютивное вычисление $comp = (D_{j1},\theta_1,G_1),(D_{j2},\theta_2,G_2),...,(D_{jk},\theta_n,G_n)$ называется

□ успешным вычислением (SLD-резолютивным опровержением), если G_n = □;

□ бесконечным вычислением, если comp—это бесконечная последовательность;

□ тупиковым вычислением , если comp — это конечная последовательность, и при этом для запроса G_n невозможно построить ни одной SLD-резольвенты.

Определение
Стратегией вычисления запросов к логическим программам называется алгоритм построения (обхода) дерева SLD-резолютивных вычислений $T_{G_0,P}$ всякого запроса G_0 к произвольной логической программе P
Стратегия вычислений называется вычислительно полной , если для любого запроса G_0 и любой логической программы P эта стратегия строит (обнаруживает) все успешные вычисления запроса G_0 к программы P

стратегия обхода в ширину , при которой дерево строится (обходится) поярусно — вершина i-го не строится, до тех пор пока не будут построены все вершины (i − 1)-го яруса; Полная, но долго.
стратегия обхода в глубину с возвратом , при которой ветви дерева обходятся поочередно — очередная ветвь дерева не обходится, до тех пор пока не будут пройдены все вершины текущей ветви. Неполная, но быстро, стандарт.

5. Транзакционное управление в СУБД. Методы сериализации транзакций.

В современных СУБД поддерживается понятие транзакции, характеризуемое аббревиатурой ACID (Atomicy, Consistency, Isolation и Durability). В соответствии с этим понятием под транзакцией разумеется последовательность операций над базой данных, обладающая следующими свойствами.

- *Атомарность (Atomicy)*. Это свойство означает, что результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных не должно быть отражено действие ни одной операции.
- *Согласованность (Consistency).* В классическом смысле это свойство означает, что транзакция может быть успешно завершена с *фиксацией* результатов своих операций только в том случае, когда действия операций не нарушают *целостность* базы данных, т.е. удовлетворяют набору ограничений целостности, определенных для этой базы данных. Это свойство расширяется тем, что во время выполнения транзакции разрешается устанавливать точки согласованности и явным образом проверять ограничения целостности.
- *Изоляция (Isolation).* Требуется, чтобы две одновременно (параллельно или квазипараллельно) выполняемые транзакции никоим образом не действовали одна на другую.
- *Долговечность (Durability).* После успешного завершения транзакции все изменения, которые были внесены в состояние базы данных операциями этой транзакции, должны гарантированно сохраняться, даже в случае сбоев аппаратуры или программного обеспечения.

Проблемы изолированности транзакций:

- Потерянные изменения – T_1 записывает данные, затем эти же данные записывает T_2 и завершается откатом. В результате при считывании T_1 транзакция не будет видеть своих данных, даже если она не завершалась. Решение – блокировать запись для двух транзакций в одно место.
- Чтение грязных данных – T_1 пишет данные, T_2 читает эти данные, затем T_1 делает откат. T_2 считала данные, которых не должно быть в БД. Решение – блокировать чтение данных, в которые пишет транзакция.
- Неповторяющиеся чтения – T_1 читает данные, затем T_2 записывает туда новое значение и завершается. T_1 на новом считывании получит другие данные. Решение – блокировать запись считываемых данных.
- Фантомы – примерно то же самое, но считывание кортежей, которых не было раньше, до завершения транзакции.

Пусть в системе одновременно выполняется некоторое множество транзакций $S = \{T_1, T_2, \ldots, T_n\}$. План (способ) выполнения набора транзакций S называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций ($T_{i1}, T_{i2}, \ldots, T_{in}$).

Сериализация транзакций – это механизм их выполнения по некоторому сериальному плану. Обеспечение такого механизма является основной функцией компонента СУБД, ответственного за управление транзакциями. Система, в которой поддерживается сериализация транзакций, обеспечивает реальную изолированность пользователей.

Основная реализационная проблема состоит в выборе метода сериализации набора транзакций, который не слишком ограничивал бы чередование их операций или реальную параллельность. Приходящим на ум тривиальным решением является действительно последовательное выполнение транзакций. Но существуют ситуации, в которых можно выполнять операторы разных транзакций в любом порядке с сохранением свойства сериальности. Примерами могут служить только читающие транзакции, а также транзакции, не конфликтующие по объектам базы данных.

Между транзакциями T_1 и T_2 могут существовать следующие виды конфликтов:

- W/W – транзакция T_2 пытается изменять объект, измененный не закончившейся транзакцией T_1 (наличие такого конфликта может привести к возникновению ситуации потерянных изменений);
- R/W – транзакция T_2 пытается изменять объект, прочитанный не закончившейся транзакцией T_1 (наличие такого конфликта может привести к возникновению ситуации неповторяющихся чтений);
- W/R – транзакция T_2 пытается читать объект, измененный не закончившейся транзакцией T_1 (наличие такого конфликта может привести к возникновению ситуации «грязного» чтения).

Практические методы сериализации транзакций основываются на учете этих конфликтов.

6. Аппаратно-программные средства поддержки мультипрограммного режима – система прерываний, защита памяти, привилегированный режим.

Выше уже речь уже шла о **мультипрограммном режиме**, когда в обработке могут находиться две и более программы пользователей, и каждая из этих программ может находиться в одном из трех состояний: во-первых, программа может выполняться на процессоре (т.е. ее команды исполняются центральным процессором), во-вторых, программа может ожидать завершения запрошенного ею обмена (для продолжения ее выполнения необходимо окончания обмена), и, наконец, в-третьих, программы могут находиться в ожидании освобождения центрального процессора (эти программы готовы к выполнению на процессоре, но процессор в данный момент занят иной программой).
Мультипрограммный режим — это режим наиболее эффективной загрузки центрального процессора. На сегодняшний день мультипрограммный режим позволяет обрабатываться на компьютере большому числу процессов (задач), предоставляющих пользователю широкий круг различных услуг.
Под **корректным** функционированием мы будем понимать, что в независимости от степени мультипрограммирования (от количества обрабатываемых в системе программ) результат работы конкретной программы не зависит от наличия и деятельности других программ. Чтобы понять, какие требования предъявляются подобным системам, разберем сначала, какие трудности и проблемы могут возникнуть при мультипрограммном режиме.
Первая проблема, которая может возникнуть, — это влияние программ друг на друга. Очень нежелательна ситуация, когда одна программа может обратиться в адресное пространство другой программы и считать оттуда данные, и уж совсем плоха ситуация, когда другая программа может что-то записать в чужое адресное пространство. Соответственно, для корректного мультипрограммирования система должна обеспечивать эксклюзивное владение программ выделенными им участками памяти. Если возникает задача обеспечения множественного доступа к памяти, то это должно осуществляться с согласия владельца этой памятью. Итак, первое требование к системе — это наличие т.н. ***аппарата защиты памяти***. Сразу отметим, что режим защиты памяти нельзя делать чисто программным способом, поскольку если данный режим будет обеспечивать операционная система (т.е. каждый раз сравнивать получаемый исполнительный адрес, не вышел ли он за границы дозволенного программе диапазона адресов), то производительность вычислительной системы в целом будет крайне низкой.
Реализация аппарата защиты памяти может быть достаточно простой: в процессоре могут быть специальные регистры (***регистры границ***), в которых устанавливаются границы диапазона доступных для исполняемой задачи адресов оперативной памяти. Соответственно, когда устройство управления в центральном процессоре вычисляет очередной исполнительный адрес (это может быть адрес следующей команды или же адрес необходимого операнда), **автоматически** проверяется, принадлежит ли полученный адрес заданному диапазону. Если адрес принадлежит диапазону, то продолжается обработка задачи, иначе же в системе возникает прерывание (т.н. ***прерывание по защите памяти***). Отметим, что предложенная модель в реальной аппаратуре может быть реализована множеством способов, но главное, что при постановке программы на обработку операционная система (программным способом) задает значения указанных регистров границ, а дальнейшая проверка адресов осуществляется аппаратным способом.
Рассмотрим следующий круг возникающих при мультипрограммном режиме проблем. Предположим, в нашей мультипрограммной системе имеется единственное печатающее устройство, и есть несколько программ, которые выводят свои данные на печать данному устройству. Соответственно, если каждая программа будет иметь доступ к командам управления конечных физических устройств, то при совместной работе в режиме мультипрограммирования эти программы будут вперемешку обращаться к печатающему устройству и печатать на нем порции своих данных, что в итоге приведет к невозможности интерпретации напечатанной информации. Система должна каким-то способом ранжировать и в соответствии с этим ранжированием ограничивать доступ пользователей различных категорий к машинным командам. Решением стала **аппаратная** возможность работы центрального процессора в двух режимах: в ***режиме работы операционной системы*** (или ***привилегированном режиме***, или ***режиме супервизора***) и в ***пользовательском режиме*** (или ***непривилегированном режиме***). В режиме работы ОС процессор исполняет абсолютно все команды, представленные в программе. Если же программа исполняется в пользовательском режиме, то ей доступны для исполнения лишь некоторое подмножество машинных команд (если же при обработке такой программы встретится недопустимая команда, то в системе возникнет прерывание по запрещенной команде).
Тогда возникает вопрос, что должна делать программа, обрабатываемая в пользовательском режиме, для печати, например, своих данных. Решений здесь может быть достаточно много, одним из которых может быть наличие в системе специальных команд, интерпретируемых как обращения к операционной системе (которые в некоторых системах рассматриваются как прерывания, в других системах — не как прерывания; мы будем рассматривать их как прерывания по обращению к операционной системе). Тогда программа, работающая в непривилегированном режиме, может вызывать команды обращения к операционной системе, а через параметры, положим, передавать необходимые данные, которые могут свидетельствовать о желании данной программы распечатать какую-то информацию на устройстве печати. Тогда схема организации печати данных на

устройстве печати может выглядеть следующим образом. Операционная система получает от пользователей (т.е. от пользовательских программ) заказы на печать, и для каждой из программы она формирует некоторую таблицу или область памяти, в которой будет аккумулироваться информация, которую необходимо вывести на принтер. Тогда каждый запрос программ на печать порции данных не является реальным обращением к устройству печати, но свидетельствует лишь о том, что передаваемая порция данных должна быть распечатана, а ОС их аккумулирует. Реальная печать будет осуществляться при возникновении одного из трех событий. Во-первых, программа, посылающая данные на печать, успешно завершилась. Это означает, что гарантированно она не будет более посылать данные на печать. Во-вторых, в программе обнаружилась фатальная ошибка, что ведет к безусловному завершению этой программы, что опять-таки гарантирует отсутствие будущих запросов данной программы на печать. И, в-третьих, операционная система может получить (от некоторого виртуального оператора, т.н. *планировщика*) команду разгрузить буфер печати данной конкретной программы.

И, наконец, еще одна серьезная проблема, которая может возникнуть при организации мультипрограммного режима, связана с тем, что в выполняемой в текущий момент программе встретилась семантическая ошибка — программа зациклилась. Соответственно, если в этом цикле не встречаются команды, которые могут привести к тем или иным прерываниям, то в этом случае вся вычислительная система «зависает»: никакие новые задачи не ставятся на счет и пр. Решение данной проблемы может быть довольно простым: необходима функция управления временем. Это означает, что операционная система должна контролировать время использования центрального процессора программами пользователя. Для этих целей компьютеру требуется **прерывание по таймеру**. Резюмируя, можно сказать, что для реализации мультипрограммного режима необходимо наличие аппарата прерываний, и этот аппарат, как минимум, должен включать в себя аппарат прерывания по таймеру. В этом случае зацикленная программа будет периодически прерываться, управление периодически будет попадать операционной системе, что даст возможность поставить на счет другую программу либо снять со счета (например, по команде пользователя) эту зависшую программу.

Итак, требуются три аппаратных средства компьютера, необходимых для поддержки мультипрограммного режима: **аппарат защиты памяти, специальные режимы исполнения команд и аппарат прерываний, состоящий, как минимум, из аппарата прерывания по таймеру**. Отметим, что специальных режимов может быть больше двух: т.е. часть команд доступна всем программам, часть команд могут выполняться лишь в защищенном режиме, еще часть — в более защищенном режиме, и т.д.

Может возникнуть резонный вопрос, как происходит включение режима супервизора. Ответ здесь будет зависеть от архитектуры конкретной системы. Например, в некоторых архитектурах считается, что операционная система занимает некоторое предопределенное адресное пространство физической памяти. И если управление попадает на эту область, то включается режим операционной системы. А вот выключение режима операционной системы может происходить программно: например, операционная система, запуская процесс, может предварительно программным способом установить его в непривилегированный режим.

7. Организация взаимодействия процессов и средства их синхронизации. Классические задачи синхронизации.

Одной из важных проблем, которые появились в современных операционных системах, является проблема взаимодействия процессов.

Будем говорить, что процессы называются ***параллельными***, если время их выполнения хотя бы частично перекрываются. Не стоит забывать, что, говоря о параллельных процессах, речь идет лишь о ***псевдопараллелизме***, поскольку реально на процессоре может исполняться только один процесс.

Параллельные процессы могут быть ***независимыми*** и ***взаимодействующими***. Независимые процессы используют множество независимых ресурсов, т.е. те ресурсы, которые принадлежат независимым процессам, в пересечении дают пустое множество. Альтернативой независимым процессам являются взаимодействующие процессы — те процессы, пересечение множеств ресурсов которых непустое. При этом одни процессы могут оказывать влияние на другие процессы, участвующие в этом взаимодействии.

Совместное использование ресурсов двумя и более процессами, когда каждый из них некоторое время владеет этими ресурсами, называется ***разделением ресурсов*** (как аппаратных, так и программных, или виртуальных). Разделяемый ресурс, использование которого организовано таким образом, что он может быть доступен в каждый момент времени только одному из взаимодействующих процессов, называется ***критическим ресурсом***. Соответственно, часть программы, в рамках которой осуществляется работа с критическим ресурсом, называется ***критической секцией***.

При организации корректного взаимодействия процессов очень важно требование, декларирующее, что результат работы взаимодействующих процессов не должен зависеть от порядка переключения выполнения между этими процессами, т.е. от соотношения скорости выполнения данного процесса со скоростями выполнения других процессов.

Гонка процессов – когда результат вычисления зависит от того, какой процесс быстрее работает.

Для минимизации проблем, возникающих при гонках, используется ***взаимное исключение*** — такой способ работы с разделяемым ресурсом, при котором в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ. Для организации модели взаимного исключения используются различные модели синхронизации. Прежде, чем рассматривать их, необходимо отметить те проблемы, которые могут возникать при организации взаимного исключения — это ***тупики*** и ***блокировки***.

Блокировка — это ситуация, когда доступ к разделяемому ресурсу одного из взаимодействующих процессов не обеспечивается за счет активности более приоритетных процессов.

Тупик, или ***deadlock***, — это ситуация «клинчевая», когда из-за некорректной организации доступа к разделяемым ресурсам происходит взаимоблокировка.

Способы организации взаимного исключения

Семафоры Дейкстры — это формальная модель организации доступа, предложенная голландским ученым Дейкстрой, которая основывается на следующей концепции. Имеется специальный тип данных — ***семафор***. Переменная типа ***семафор*** может иметь целочисленные значения. Над этими переменными определены следующие операции: **down(**S**)** (или **P(**S**)**) и **up(**S**)** (или **V(**S**)**).

Оригинальные обозначения **P** и **V**, данные Дейкстрой и получившие широкое распространение в литературе, являются сокращениями голландских слов *proberen* — проверить и *verhogen* — увеличить.

Операция **down(S)** проверяет значение семафора **S**, и если оно больше нуля, то уменьшает его на **1**. Если же это не так, процесс блокируется, причем связанная с заблокированным процессом операция **down** считается незавершенной.

Операция **up(S)** увеличивает значение семафора на **1**. При этом, если в системе присутствуют процессы, блокированные ранее при выполнении **down** на этом семафоре, один из них разблокировывается и завершает выполнение операции **down**, т.е. вновь уменьшает значение семафора. Выбор процесса никак не оговаривается.

При этом операции **up** и **down** являются атомарными (неделимыми), т.е. их выполнение не может быть прервано прерыванием.

Двоичный семафор — семафор, максимальное значение которого равно 1. Этот тип семафоров обеспечивает взаимное исключение.

Заметим, что требование атомарности операций **down** и **up** накладывает ограничения на реализацию семафоров Дейкстры, и зачастую это сложная задача. Существуют программные реализации, но в них атомарность не всегда присутствует.

Мониторы Хоара — модель синхронизации, в которой, в частности, предпринята попытка обойти требование аппаратной поддержки атомарности упомянутых выше операций. Монитор является высокоуровневой конструкцией (можно говорить, что это конструкция уровня языка программирования), реализация которой поддерживается системой программирования (компилятором). Монитор — это специализированный модуль, включающий в себя некие процедуры и функции, а также данные, с которыми работают эти процедуры и функции. При этом данный модуль обладает следующими свойствами:

- данные монитора доступны только через процедуры и функции этого монитора;

- считается, что процесс занимает (или входит) монитор тогда, когда он начинает использовать одну из процедур или функций монитора;
- в любой момент времени внутри монитора может находиться не более одного процесса, остальные процессы в зависимости от используемой стратегии поведения либо получает отказ, либо блокируется, становясь в очередь.

Иллюстрацией монитора может служить кабина таксофонного аппарата.
Повторим, что монитор — это языковая конструкция с централизованным управлением (в отличие от семафоров, которые не обладают централизацией). Семафоры и мониторы являются средствами организации работы в основном в однопроцессорных системах либо многопроцессорных системах с общей памятью. В многопроцессорных системах с распределенной памятью эти средства не очень подходят. Для них в настоящий момент часто используется механизм ***передачи сообщений***.
Механизм ***передачи сообщений*** основан на двух функциональных примитивах: **send** (отправить сообщение) и **receive** (принять сообщение). Данные операции можно разделить по трем критериям: синхронизация, адресация и длина сообщения.
Синхронизация. Операции посылки/приема сообщений могут быть ***блокирующими*** и ***неблокирующими***. Рассмотрим различные комбинации.
Блокирующий send: процесс-отправитель будет блокирован до тех пор, пока посланное им сообщение не будет получено.
Блокирующий receive: процесс-получатель будет блокирован до тех пор, пока не будет получено соответствующее сообщение.
Соответственно, неблокирующие операции, как следует из названия, происходят без блокировок.
Итак, комбинируя различные операции send и receive, мы получаем 4 различных модели синхронизации. Отметим одно важное свойство аппарата сообщений, заключающееся в том, что в нем явно совмещены средства передачи информации и синхронизации, таким образом, механизм передачи сообщений можно использовать для достижения двух целей.
Адресация может быть ***прямой***, когда указывается конкретный адрес получателя и/или отправителя (например, когда получатель ожидает сообщения от конкретного отправителя, игнорируя сообщения других отправителей), или ***косвенной***. В случае косвенной адресации не указывается адрес получателя при отправке или отправителя при получении; сообщение «бросается» в некоторый общий пул, в котором могут быть реализованы различные стратегии доступа (FIFO, LIFO и т.д.). Этим пулом может выступать очередь сообщений (FIFO) или почтовый ящик, в котором может быть реализована любая модель доступа.
Итак, повторимся, что данный механизм совмещает два средства: средство передачи данных и синхронизации. Этот аппарат является базовым средством организации взаимодействия процессов в многопроцессорных системах с распределенной памятью.

Классические задачи синхронизации процессов

Классические задачи синхронизации процессов отражают разные модели взаимодействия и демонстрируют использование механизма семафоров для организации такого взаимодействия.
Обедающие философы. Пускай существует круглый стол, за которым сидит группа философов: они пришли пообщаться и покушать. Кушают они спагетти, которое находится в общей миске, стоящей в центре стола. Для приема пищи они пользуются двумя вилками: одна в левой руке, другая — в правой. Вилки располагаются по одной между каждыми двумя философами. Любой философ может взять обе вилки, покушать, затем положить вилки на стол, после этого вилки может взять его сосед и повторить эти действия. Если мы организуем работу таким образом, что любой философ, желающий поесть, берет сначала левую вилку, затем правую, после чего начинает кушать, то в какой-то момент может возникнуть ситуация тупика (когда каждый возьмет по одной левой вилке, а правая будет захвачена соседом).

Обещающие философы.
Итак, данная задача иллюстрирует модель доступа равноправных процессов к общему ресурсу, и ставится вопрос, как организовать **корректную** работу такой системы.
В этом решении каждый философ живет по аналогичному циклическому распорядку: размышляет некоторое время, затем берет вилки, кушает, кладет вилки. Рассмотрим процедуру получения вилок (TakeForks). Опускается семафор mutex, который используется для синхронизации входа в критическую секцию. Внутри критической секции меняем состояние философа (помечаем его состоянием «голоден»). Затем предпринимается попытка начать есть (вызывается функция Test). Функция Test проверяет, что если i-ый философ голоден, а его соседи в данный момент не едят (т.е. правая и левая вилки свободны), то этот философ начинает прием пищи (состояние EATING), а его семафор поднимается (заметим, что изначально этот семафор инициализирован нулем). После этого мы возвращаемся обратно в функцию TakeForks, в которой далее происходит выход из критической секции (подымаем семафор mutex), а затем опускаем семафор этого философа. Если внутри функции Test философу удалось начать прием пищи, то семафор поднят, и операция down обнулит его, не блокируясь. Если же функция Test не изменит состояние философа, а также не поднимет его семафор, то операция down в этой точке заблокируется до тех пор, пока оба соседа не освободят вилки.

Внутри функции освобождения вилок PutForks первым делом происходит опускание семафора mutex, происходит вход в критическую секцию. Затем меняется статус философа (на статус THINKING), после чего проверяем его соседей: если любой из них был заблокирован лишь из-за того, что наш i-ый философ забрал его вилку, то мы его разблокируем, и он начинает прием пищи. После этого происходит выход из критической секции путем подъема семафора mutex.
Заметим, что использование механизма взаимоисключающего нахождения внутри критической секции (за счет семафора mutex) гарантирует, что не возникнет ситуация, когда два процесса, соответствующие соседним философам, будут так спланированы на обработку на процессоре, что функция Test в каждом из них проработает и разрешит каждому из них начать прием пищи (что, конечно же, является ошибкой). Если же этого механизма не будет, то возможно, что один их процессов-соседей входит в Test, делает проверку на возможность начала приема пищи. Проверка дает истинное значение, управление переходит к первой команде внутри if-блока. После этого происходит смена процесса на процессоре, управление получает сосед этого философа. Тот тоже делает проверку внутри функции Test, и также получает положительный ответ, и управление переходит к первой инструкции if-блока. Дальнейшая работа будет некорректной.
Задача «читателей и писателей». Представим произвольную систему резервирования ресурса. Например, это может быть система резервирования места в гостинице. В данной системе существует два типа процессов для работы с информацией. Одни процессы могут читать информацию, а другие — ее изменять, корректировать. Соответственно, возникает все тот же вопрос, как организовать корректную совместную работу этих процессов. Это означает, что в любой момент времени читать данные могут любое количество процессов-читателей, но если процесс-писатель начал свою работу, то все остальные процессы (и читатели, и писатели) будут блокированы на входе в систему.
Рассмотрим модельную реализацию данной задачи при выбранной следующей стратегии: будем считать, что наиболее приоритетными являются читающие процессы. Т.е. процесс-писатель будет ожидать момента, когда все желающие процессы-читатели окончат свои действия в системе и покинут ее.
В приведенном решении процесс-читатель в каждом цикле своей работы входит в критическую секцию (за счет опускания семафора mutex), увеличивает счетчик читателей, находящихся в хранилище, на 1. Затем проверяет, что если он является первым читателем (т.е. в данный момент он единственный клиент в хранилище), то опускает семафор db, тем самым, препятствуя писателем войти в систему, если они того пожелают. Если же семафор db уже был опущен, то это означает, что в данный момент в хранилище присутствует писатель, и этот первый читатель заблокируется на этой операции, ожидая выхода писателя из системы. (Заметим, что это блокировка происходит внутри критической секции, поэтому остальные читатели будут блокироваться на опускании семафора mutex.) После этого происходит выход из критической секции (подымаем семафор mutex), чтение информации из хранилища. Затем производятся обратные действия по выходу из хранилища, которые также происходят внутри критической секции. Итак, на выходе мы уменьшаем число читателей в хранилище, и если этот читатель является последним клиентом в библиотеке, то происходит поднятие семафора db, разрешая работу писателям (которые к этому моменту могли быть заблокированы на входе). В конце цикла работы читатель обрабатывает полученные данные из хранилища, после чего цикл повторяется.
Писатель в начале каждого цикла своей работы подготавливает данные для сохранения, затем пытается войти в хранилище, опуская семафор db. Если в хранилище кто-то есть, то он будет ожидать, пока последний клиент (независимо, читатель это или писатель) не покинет его. После этого он производит корректировку данных в хранилище и покидает его, поднимая семафор db.
Заметим, что в данном решении если хотя бы один читатель находится внутри системы, то любой следующий читатель беспрепятственно в нее попадет, писатель же будет ожидать, когда все посетители покинут хранилище, т.е. реализована стратегия приоритетности читателя перед писателем.
Данная задача иллюстрирует модель доступа к общему ресурсу процессов, имеющих разные приоритеты.
Задача о «спящем парикмахере». Представим себе парикмахерскую, в которой имеется единственно рабочее кресло и единственный цирюльник. В парикмахерской есть комната ожидания, в которой стоят N кресел, на которых могут сидеть клиенты, ожидающие своей очереди. Если свободных кресел нет, то вновь приходящие клиенты сразу же покидают заведение. Когда посетителей нет, парикмахер может сидеть в своем кресле и дремать.
Данная задача является иллюстрацией модели клиент-сервер с ограничением на длину очереди клиентов.
Рассмотрим реализацию данной модели.
Процесс-парикмахер первым делом опускает семафор customers, уменьшив тем самым количество ожидающих посетителей на 1. Если в комнате ожидания никого нет, то он «засыпает» в своем кресле, пока не появится клиент, который его разбудит. Затем парикмахер входит в критическую секцию, уменьшает счетчик ожидающих клиентов, поднимает семафор barbers, сигнализируя клиенту о своей готовности его обслужить, а потом выходит из критической секции. После описанных действий он начинает стричь волосы посетителю.

Посетитель парикмахерской входит в критическую секцию. Находясь в ней, он первым делом проверяет, есть ли свободные места в зале ожидания. Если нет, то он просто уходит (покидает критическую секцию, поднимая семафор mutex). Иначе он увеличивает счетчик ожидающих процессов и поднимает семафор customers. Если же этот посетитель является единственным в данный момент клиентом брадобрея, то он этим действием разбудит брадобрея. После этого он выходит из критической секции и «захватывает» брадобрея (опуская семафор barbers). Если же этот семафор опущен, то клиент будет дожидаться, когда брадобрей его поднимет, известив тем самым, что готов к работе. В конце клиент обслуживается (GetHaircut).

8. Виртуальная память. Модели организации оперативной памяти.

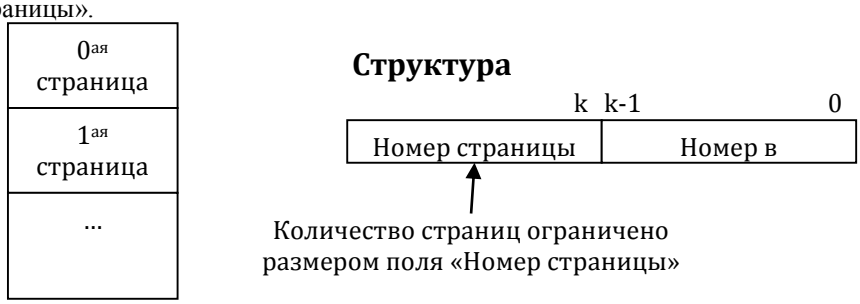
Следующий аппарат компьютера, который также сильно связан с поддержкой программного обеспечения, — это аппарат ***виртуальной памяти***. Что понимается под виртуальной памятью и виртуальным адресным пространством? Неформально виртуальное адресное пространство можно определить как то адресное пространство, которое используется внутри программ (написанных, например, на языках программирования высокого уровня). Виртуальные адреса существуют «вне машины». Соответственно, стоит проблема привязки виртуального адресного пространства физической памяти. И эта проблема решается за счет аппарата виртуальной памяти.

Итак, ***аппарат виртуальной памяти*** — это аппаратные средства компьютера, обеспечивающие преобразование (установление соответствия) программных адресов, используемых в программе, адресам физической памяти, в которой размещена программа при выполнении. И реализацией одной из моделей аппарата виртуальной памяти является аппарат ***базирования адресов***.

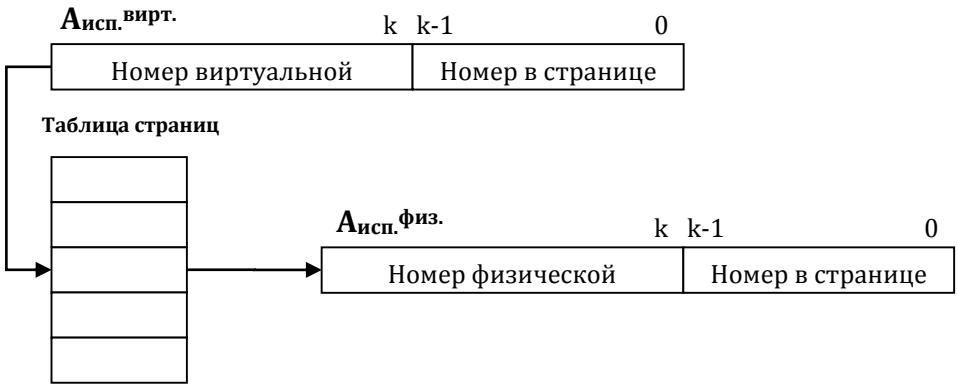
Механизм базирования адресов основан на двоякой интерпретации получаемых в ходе выполнения программы исполнительных адресов ($А_{исп.}^{прог.}$). С одной стороны, его можно интерпретировать как ***абсолютный исполнительный адрес***, когда физический адрес в некотором смысле соответствует исполнительному адресу программы ($А_{исп.}^{физ.} = А_{исп.}^{прог.}$). Например, требуется «прочитать ячейку с адресом (абсолютным адресом) 0», или «передать управление по адресу входа в обработчик прерывания». С другой стороны, исполнительный адрес программы можно проинтерпретировать как ***относительный адрес,*** т.е. адрес, зависящий от места дислокации программы в ОЗУ. Иными словами, имеется оперативная память с ячейками с номерами от 0 до некоторого A–1, и, начиная с некоторого адреса K, расположена программа. Тогда адрес $А_{исп.}^{прог.}$ внутри программы можно трактовать, как отступ от физической ячейки с адресом K на величину $А_{исп.}^{прог.}$. Для реализации модели базирования используется специальный ***регистр базы***, в который в момент загрузки процесса в оперативную память операционная система записывает начальный адрес загрузки (т.е. K). Тогда реальный физический адрес получается, исходя из формулы $А_{исп.}^{физ.} = А_{исп.}^{прог.} + <R_{базы}>$.

Аппарат базирования позволяет разрешить проблему перемещаемости программ по ОЗУ, поскольку процесс можно загрузить в любую область памяти. Но при этом необходимо помнить, что программа представляется в виде непрерывной области виртуальной памяти, которая загружается в непрерывный фрагмент физической памяти.

Развитием аппарата виртуальной памяти является аппарат ***страничной организации памяти***. Ниже мы рассмотрим модельный сильно упрощенный пример страничной памяти. Данная модель представляет все адресное пространство оперативной памяти в виде последовательности страниц. ***Страница*** — это область адресного пространства фиксированного размера: обычно размер страницы кратен степени двойки, будем считать, что размер страницы 2^k. Тогда все адресное пространство представимо в виде последовательности страниц (нулевая, первая и т.д.). Сказанное означает, что структура адреса представима в виде двух полей (**Ошибка! Источник ссылки не найден.**): правые k разрядов представляют адрес внутри страницы, а оставшиеся разряды отвечают за номер страницы. Тогда количество страниц в системе ограничено разрядностью поля «Номер страницы».



В центральном процессоре имеется аппаратная таблица, называемая ***таблицей страниц***, предназначенная для следующих целей. Количество строк в этой таблице определяется максимальным числом виртуальных страниц, ограниченное схемами работы процессора и максимальной адресной разрядностью процессора. Каждой виртуальной странице ставится в соответствие строка таблицы страниц с тем же номером (нулевой странице соответствует нулевая строка, и т.п.). Внутри каждой записи таблицы страниц находится номер физической страницы, в которой размещается соответствующая виртуальная страница программы. Соответственно, аппарат виртуальной страничной памяти позволяет **автоматически** (т.е. **аппаратно**) преобразовывать номер виртуальной страницы на номер физической страницы посредством обращения к таблице страниц. Программных действий при таком подходе требуется минимально: при выборе операционной системой очередного процесса, который ставится на обработку на центральный процессор, она должна лишь корректно заполнить аппаратную таблицу страниц процессора для данного процесса.



Страничная организация памяти. Преобразование виртуального адреса в физический.
Типовая схема преобразования адресов достаточно проста. Пускай в таблице страниц имеется N строк. Это означает, что в компьютере дозволено использовать N страниц. Содержимое каждой i-ой строки таблицы — α_i, оно определяется операционной системой в момент запуска процесса. Пускай в нашем модельном примере если $\alpha_i >= 0$, то это номер физической страницы, которая соответствует i-ой виртуальной странице. Если $\alpha_i < 0$, то это означает, что данной страницы у программы нет, и если в ходе обработки процесса процессор обращается к строке таблицы страниц с отрицательным содержимым, происходит прерывание по защите памяти. Причин возникновения прерывания в данном случае может две. Во-первых, может оказаться, что действительно i-ой виртуальной страницы у программы нет, что свидетельствует об ошибке в программе. Во-вторых, может оказаться, что соответствующей страницы нет в оперативной памяти, она расположена на внешнем запоминающем устройстве (ВЗУ), т.е. данная i-ая виртуальная страница легальна, но в данный момент ее нет в ОЗУ. Так или иначе, операционная система анализирует причину возникновения прерывания и для последнего случая осуществляет подкачку из ВЗУ в ОЗУ требуемой страницы.
Отметим, что страничная организация памяти решает все вышеперечисленные проблемы, связанные с выполнением программ. Здесь имеется механизм защиты памяти (в этой схеме процесс никогда не сможет обратиться к «чужой» странице), но также имеется возможность разделять некоторые страницы между несколькими процессами (в этом случае операционная система каждому из процессов допишет в таблицу страниц номер общей страницы). Данная схема обладает достаточной производительностью, поскольку ее функционирование построено на использовании регистров. Также данный подход решает проблему фрагментации, поскольку все программы оперируют в терминах страниц (каждая из которых имеет фиксированный размер). Помимо этого решается еще и проблема перемещаемости программ по ОЗУ, причем даже в рамках одной программы соответствие между виртуальными и физическими страницами может оказаться произвольным: ее нулевая виртуальная страница может располагаться в одной физической странице, первая виртуальная — в другой (совершенно не связанной с первой) физической странице, и т.д. Еще одним важным достоинством страничной организации памяти заключается в том, что нет необходимости держать в оперативной памяти весь исполняемый процесс. Реально в ОЗУ может находиться лишь незначительное число страниц, в которых расположены команды и требуемые для текущих вычислений операндов, а все оставшиеся страницы могут находиться на внешней памяти — в областях подкачки. Как следствие только что сказанного является то, что размеры физической и виртуальной памяти могут быть произвольными. Может оказаться, что физической памяти в компьютере больше, чем размеры адресного пространства виртуальной памяти, а может оказаться и наоборот: физической памяти существенно меньше виртуальной. Но во всех этих случаях система окажется работоспособной.
Но данный подход имеет и свои недостатки. Во-первых, это страничная фрагментация, или ***внутренняя*** (скрытая) ***фрагментация***: если в странице используется хотя бы один байт, то вся страница отводится процессу (т.е., решив вопрос с т.н. ***внешней фрагментацией***, в указанном случае не используется память, размером со страницу минус один байт). К тому же описанная выше модель является вырожденной: если таблица страниц целиком располагается на регистровой памяти, то в силу дороговизны последней размеры подобной таблицы будут слишком малы (а следовательно, будет невелико количество физических страниц). Реальные современные системы имеют более сложную логическую организацию, и речь о ней пойдет ниже.
Напоследок заметим, что данное нами определение аппарата виртуальной памяти расходится с определениями некоторых других источников. Повторим, что мы рассматриваем механизм виртуальной памяти как механизм преобразования виртуального адресного пространства в физическое. Во многих изданиях, посвященных рассмотрению операционных систем, виртуальной памятью считается то, что позволяет часть программы размещать на внешних устройствах, т.е. считают механизм виртуальной памяти как средство увеличения объема физической памяти. Мы считаем такое определение некорректным. Если рассматривать, например, виртуальную память как механизм увеличения объема, то возникает вопрос: в случае большого объема физической памяти разве виртуальная память отсутствует? Соответственно, возникают проблемы с подобным определением.

9. Алгоритм Сети-Ульмана оптимального распределения регистров и его обоснование.

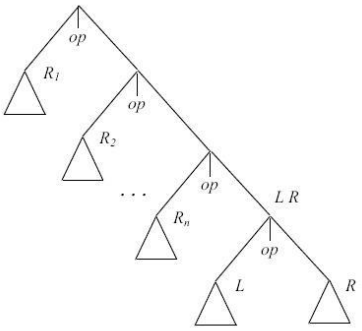
Задача оптимального распределения регистров.

Одной из важнейших задач при генерации кода является распределение регистров. Рассмотрим хорошо известную технику распределения регистров при трансляции арифметических выражений, называемую алгоритмом Сети-Ульмана.

Условия применения алгоритма.

Пусть система команд машины имеет неограниченное число универсальных регистров, в которых выполняются арифметические команды. Рассмотрим, как можно сгенерировать код, используя для данного арифметического выражения минимальное число регистров.

Пусть имеется синтаксическое дерево выражения. Предположим сначала, что распределение регистров осуществляется по простейшей схеме сверху-вниз слеванаправо. Тогда к моменту генерации кода для поддерева LR занято n регистров. Пусть поддерево L требует n_l регистров, а поддерево R – n_r регистров. Если $n_l = n_r$, то при вычислении L будет использовано n_l регистров и под результат будет занят (n+1)-й регистр. Еще $n_r(= n_l)$ регистров будет использовано при вычислении R. Таким образом, общее число использованных регистров будет равно n + nl + 1. Если $n_l > n_r$, то при вычислении L будет использовано n_l регистров. При вычислении R будет использовано $n_r < n_l$ регистров, и всего будет использовано не более чем $n + n_l$ регистров. Если $n_l < n_r$, то после вычисления L под результат будет занят один регистр (предположим, (n + 1)-й) и nr регистров будет использовано для вычисления R. Всего будет использовано $n + n_r + 1$ регистров.



Обоснование алгоритма (минимальность числа используемых регистров при принятых ограничениях).

Видно, что для деревьев, совпадающих с точностью до порядка потомков каждой вершины, минимальное число регистров при распределении их слева-направо достигается на дереве, у которого в каждой вершине слева расположено более "сложное" поддерево, требующее большего числа регистров. Таким образом, если дерево таково, что в каждой внутренней вершине правое поддерево требует меньшего числа регистров, чем левое, то, обходя дерево слева направо, можно оптимально распределить регистры. Без перестройки дерева это означает, что если в некоторой вершине дерева справа расположено более сложное поддерево, то сначала сгенерируем код для него, а затем уже для левого поддерева.

Разметка дерева.

1) если вершина – правый лист или дерево состоит из единственной вершины, помечаем эту вершину числом 1, если вершина – левый лист, помечаем ее 0
2) если вершина имеет прямых потомков с метками l_1 и l_2, то в качестве метки этой вершины выбираем наибольшее из чисел l_1 или l_2 либо число $l_1 + 1$, если $l_1 = l_2$

Распределение регистров.

1) Корню назначается первый регистр
2) Если метка левого потомка меньше метки правого, то левому потомку назначается регистр на единицу больший, чем предку, а правому - с тем же номером (сначала вычисляется правое поддерево и его результат помещается в регистр R), так что регистры занимаются последовательно. Если же метка левого потомка больше или равна метке правого потомка, то наоборот, правому потомку назначается регистр на единицу больший, чем предку, а левому - с тем же номером (сначала вычисляется левое поддерево и его результат помещается в регистр R)

Генерация кода.

1) если вершина - правый лист с меткой 1, то ей соответствует код `MOVE X, R` где R - регистр, назначенный этой вершине, а X - адрес переменной, связанной с вершиной
2) если вершина внутренняя и ее левый потомок - лист с меткой 0, то ей соответствует код

```
Код правого поддерева
Op X, R
```

где R - регистр, назначенный этой вершине, X - адрес переменной, связанной с вершиной, а Op - операция, примененная в вершине

3) если непосредственные потомки вершины не листья и метка правой вершины больше или равна метки левой, то вершине соответствует код

```
Код правого поддерева
Код левого поддерева
Op R+1, R
```

где R - регистр, назначенный внутренней вершине, и операция Op, вообще говоря, не коммутативная

4) если непосредственные потомки вершины не листья и метка правой вершины меньше метки левой вершины, то вершине соответствует код

```
Код левого поддерева
Код правого поддерева
Op R, R+1
MOVE R+1, R
```

10. Основные принципы объектно-ориентированного программирования.

Центральной идеей ООП является реализация понятия "**абстракция"**. Смысл абстракции заключается в том, что *сущность* произвольной сложности можно рассматривать, а также производить определенные действия над ней, как над **единым целым**, не вдаваясь в детали внутреннего построения и функционирования.

При создании программного комплекса необходимо разработать определенные абстракции. Одним из основных способов создания абстракции является использование концепции **иерархической классификации**. Ее суть заключается в том, что сложные системы разбиваются на более простые фрагменты. Практически все сложные системы иерархичны, и уровни их иерархии отражают различные уровни абстракции. Для каждой конкретной задачи рассматривается соответствующий уровень. Выбор низшего уровня абстракции достаточно произволен. Выбранный уровень в одном случае в качестве низшего уровня может оказаться уровнем достаточно высокой абстракции в другом проекте.

Различают **типовую** иерархию и **структурную** иерархию, которые далее мы будем называть соответственно **структурой классов** и **структурой объектов.**

Во всех объектно-ориентированных языках программирования реализованы следующие **основные механизмы (постулаты) ООП**:

- **Инкапсуляция**
- **Наследование**
- **Полиморфизм**

Все эти механизмы важны для разработки и использования абстракций**.**

1) Инкапсуляция – механизм, связывающий вместе код и данные, которыми он манипулирует, и одновременно защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому. Доступ к коду и данным жестко контролируется интерфейсом.

Основой инкапсуляции при ООП является **класс.**

Механизм инкапсуляции позволяет оставлять скрытыми от пользователя некоторые детали реализации класса (то есть инкапсулировать их в классе), что упрощает работу с объектами этого класса.

2) Наследование – механизм, с помощью которого один объект (производного класса) приобретает свойства другого объекта (родительского, базового класса). При использовании наследования новый объект не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста. Наследование позволяет какому-либо объекту наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Наследование есть очень важное понятие, поддерживающее концепцию иерархической классификации.

3) Полиморфизм – механизм, позволяющий использовать один и тот же интерфейс для общего класса действий**.**

Общая концепция полиморфизма: **один интерфейс – много методов.**

Выбор конкретного действия (метода) применительно к конкретной ситуации возлагается на компилятор. Программисту же достаточно запомнить и применить один интерфейс, вместо нескольких, что также упрощает работу.

Различаются **статический** (реализуется на этапе компиляции с помощью перегрузки функций и операций), **динамический** (реализуется во время выполнения программы с помощью механизма виртуальных функций) и **параметрический** (реализуется на этапе компиляции с использованием механизма шаблонов) полиморфизм.

Примечание. Рассмотренные понятия абстракции, инкапсуляции, наследования, полиморфизма присущи не только парадигме ООП. Так, выполнение арифметических операций над целыми числами и числами с плавающей точкой осуществляются в процессоре по разным алгоритмам. Однако в данном случае полиморфизм проявляется неявно.

11. Основные этапы компиляции (лексический анализ, синтаксический анализ, семантический анализ, генерация кода и т.д.)

На начальной фазе **лексического анализа** входная программа, представляющая собой поток литер, разбивается на лексемы – слова в соответствии с определениями языка. Основными формализмами, лежащим в основе реализации лексических анализаторов, являются конечные автоматы и регулярные выражения. Лексический анализатор может работать в двух основных режимах: либо как подпрограмма, вызываемая синтаксическим анализатором для получения очередной лексемы, либо как полный проход, результатом которого является файл лексем.
В процессе выделения лексем лексический анализатор может как самостоятельно строить таблицы объектов (чисел, строк, идентификаторов и так далее), так и выдавать значения для каждой лексемы при очередном к нему обращении. В этом случае таблицы объектов строятся в последующих фазах (например, в процессе синтаксического анализа).
На этапе лексического анализа обнаруживаются некоторые (простейшие) ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и другие).

Центральная задача **синтаксического анализа** – разбор структуры программы. Как правило, под структурой понимается дерево, соответствующее разбору в контекстносвободной грамматике языка. В настоящее время чаще всего используется либо LL(1)-анализ (и его вариант – рекурсивный спуск), либо LR(1)- анализ и его варианты (LR(0), SLR(1), LALR(1) и другие). Рекурсивный спуск чаще используется при ручном программировании синтаксического анализатора, LR(1) – при использовании систем автоматического построения синтаксических анализаторов.
Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицы объектов. Ошибки, связанные со структурой программы, также обнаруживаются в процессе синтаксического анализа.

На этапе **контекстного анализа** выявляются зависимости между частями программы, которые не могут быть описаны контекстно-свободным синтаксисом. Это, в основном, связи "описание-использование", в частности, анализ типов объектов, анализ областей видимости, соответствие параметров, метки и другие. В процессе контекстного анализа таблицы объектов пополняются информацией об описаниях (свойствах) объектов.
Основным формализмом, используемым при контекстном анализе, является аппарат атрибутных грамматик. Результатом контекстного анализа является атрибутированное дерево программы. Информация об объектах может быть как рассредоточена в самом дереве, так и сосредоточена в отдельных таблицах объектов. В процессе контекстного анализа также могут быть обнаружены ошибки, связанные с неправильным использованием объектов.
Затем программа может быть переведена во внутреннее представление. Это делается для целей оптимизации и/или удобства генерации кода. Еще одной целью преобразования программы во внутреннее представление является желание иметь переносимый компилятор. Тогда только последняя фаза (генерация кода) является машинно-зависимой. В качестве внутреннего представления может использоваться префиксная или постфиксная запись, ориентированный граф, тройки, четверки и другие способы.
Фаз оптимизации может быть несколько. Оптимизации обычно делят на машинно-зависимые и машинно-независимые, локальные и глобальные. Определенная часть машинно-зависимой оптимизации выполняется на фазе генерации кода. Глобальная оптимизация пытается принять во внимание структуру всей программы, локальная – только небольших ее фрагментов. Глобальная оптимизация основывается на глобальном потоковом анализе, который выполняется на графе программы и представляет по существу преобразование этого графа. При этом могут учитываться такие свойства программы, как межпроцедурный анализ, межмодульный анализ, анализ областей жизни переменных и так далее.
Наконец, **генерация кода** – последняя фаза трансляции. Результатом ее является либо ассемблерный модуль, либо объектный (или загрузочный) модуль. В процессе генерации кода могут выполняться некоторые локальные оптимизации, такие как распределение регистров, выбор длинных или коротких переходов, учет стоимости команд при выборе конкретной последовательности команд. Для генерации кода разработаны различные методы, такие как таблицы решений, сопоставление образцов, включающее динамическое программирование, различные синтаксические методы. Конечно, те или иные фазы транслятора могут либо отсутствовать совсем, либо объединяться. В простейшем случае однопроходного транслятора нет явной фазы генерации промежуточного представления и оптимизации, остальные фазы объединены в одну, причем нет и явно построенного синтаксического дерева.

12. Построение детерминированного конечного автомата по регулярному выражению.

Определение регулярного множества.

Регулярное множество в алфавите T определяется рекурсивно следующим образом:

1) (пустое множество) - регулярное множество в алфавите T;
2) {e} - регулярное множество в алфавите T (e - пустая цепочка);
3) {a} - регулярное множество в алфавите T для каждого $\alpha \in T$;
4) если P и Q - регулярные множества в алфавите T, то регулярными являются и множества
   a. $P \cup Q$
   b. $PQ\ \{pq |\ p \in P, q \in Q\ \}$
   c. $P^* = \bigcup_{n=0}^{\infty} P^n$
5) ничто другое не является регулярным множеством в алфавите T.

Определение регулярного выражения (РВ).

Регулярное выражение в алфавите T и обозначаемое им регулярное множество в алфавите T определяются рекурсивно следующим образом:

1) (пусто) – регулярное выражение, обозначающее регулярное множество (пусто);
2) {e} - регулярное выражение, обозначающее регулярное множество {e};
3) {a} - регулярное выражение, обозначающее регулярное множество {a};
4) если p и q - регулярные выражения, обозначающие регулярные множества P и Q соответственно, то
   a. (p|q) - регулярное выражение, обозначающее регулярное множество $P \cup Q$
   b. (pq) - регулярное выражение, обозначающее регулярное множество PQ,
   c. (p*) - регулярное выражение, обозначающее регулярное множество P*;
5) ничто другое не является регулярным выражением в алфавите T.

Определение недетерминированного конечного автомата (НКА).

Недетерминированный конечный автомат (НКА) – по определению есть пятерка M = (Q, T, D, q_0, F), где

1) Q - конечное множество состояний,
2) T - конечное множество допустимых входных символов (входной алфавит),
3) D - функция переходов (отображающая множество $Q \times (T \cup \{e\})$ во множество подмножеств множества Q), определяющая поведение управляющего устройства,
4) $q_0 \in Q$ - начальное состояние управляющего устройства,
5) $F \subseteq Q$ - множество заключительных состояний.

Определение детерминированного конечного автомата (ДКА).

Пусть M = (Q, T, D, q_0, F) – НКА. Будем называть M детерминированным конечным автоматом (ДКА), если выполнены следующие два условия:

1. D(q,e) = (пусто), для любого $q \in Q$
2. D(q, a) содержит не более одного элемента для любых $q \in Q$ и $a \in T$.

Алгоритм может быть представлен либо как последовательность двух алгоритмов: НКА –> ДКА, либо непосредственно построение ДКА по дереву РВ.

НКА -> ДКА

Каждое состояние ДКА – множество состояний НКА.

Добавлять в рабочее множество вершины нового ДКА построенные, как совокупность вершин НКА, в которые можно попасть из текущего рассматриваемого множества по данному выражению.

РВ -> ДКА

Вход. Регулярное выражение r в алфавите T.

Выход. ДКА M = (Q, T, D, q_0, F), такой что L(M) = L(r).

Метод. Состояния ДКА соответствуют множествам позиций. Вначале Q и D пусты. Выполнить шаги 1-6:

(1) Построить синтаксическое дерево для пополненного регулярного выражения (r)#.

(2) Обходя синтаксическое дерево, вычислить значения функций nullable, firstpos, lastpos и followpos.

(3) Определить q_0 = firstpos(root), где root – корень синтаксического дерева.

(4) Добавить q_0 в Q как непомеченное состояние.

(5) Выполнить следующую процедуру:

```
while (в Q есть непомеченное состояние R){
  пометить R
  for (каждый символ a ∈ T, что в R имеется позиция, которой соотв. a){
        (пусть a соотв. Позициям p_1, ..., p_n,  S = ⋃_{1≤i≤n} followpos(p_i))
        if (S ≠ ∅){
        if (S ∉ Q)
              (добавить S в Q как непомеченное состояние)
        D(R, a) = S
```

```
                }
            }
        }
```

(6) Определить F как множество всех состояний из Q, содержащих позиции, связанные с символом #.

13. Построение канонической системы множеств LR(1) ситуаций и таблиц действий и переходов для LR(1) грамматик.

Правосторонний вывод.

Пусть G = (N, T, P, S) – КС-грамматика. Введем несколько важных понятий и определений.

Вывод, в котором в любой сентенциальной форме на каждом шаге делается подстановка самого правого нетерминала, называется правосторонним. Если S => u в процессе правостороннего вывода, то u – правая сентенциальная форма.

Восстановление правостороннего вывода.

Упорядоченное помеченное дерево D называется деревом вывода (или деревом разбора) цепочки w в КС-грамматике G = (N, T, P, S), если выполнены следующие условия:

(1) корень дерева D помечен S;

(2) каждый лист помечен либо $\alpha \in T$, либо e;

(3) каждая внутренняя вершина помечена нетерминалом $A \in N$;

(4) если X – нетерминал, которым помечена внутренняя вершина и X_1, ... , X_n – метки ее прямых потомков в указанном порядке, то X -> X_1 ... X_k – правило из множества P;

(5) Цепочка, составленная из выписанных слева направо меток листьев, равна w.

Процесс определения принадлежности данной строки языку, порождаемому данной грамматикой, и, в случае указанной принадлежности, построение дерева разбора для этой строки, называется синтаксическим анализом. Можно говорить о восстановлении дерева вывода (в частности, правостороннего или левостороннего) для строки, принадлежащей языку. По восстановленному выводу можно строить дерево разбора.

Схема работы магазинного анализатора, таблицы действий и переходов, конфигурации.

Автомат с магазинной памятью (МП-автомат) – это семерка M = (Q, T, Г, D, q_0, Z_0, F), где

1) Q – конечное множество состояний, представляющих всевозможные состояния управляющего устройства;
2) T – конечный входной алфавит;
3) Г – конечный алфавит магазинных символов;
4) D – отображение множества Q x (T {e}) x Г в множество конечных подмножеств Q x Г*, называемое функцией переходов;
5) $q_n \in Q$ – начальное состояние управляющего устройства;
6) $Z_0 \in$ Г – символ, находящийся в магазине в начальный момент (начальный символ магазина);
7) $F \subseteq Q$ – множество заключительных состояний

Конфигурация МП-автомата – это тройка (q, w, u), где

(1) $q \in Q$ – текущее состояние управляющего устройства;

(2) $w \in T^*$ – непрочитанная часть входной цепочки; первый символ цепочки w находится под входной головкой; если w = e, то считается, что вся входная лента прочитана;

(3) $u \in$ Г* – содержимое магазина; самый левый символ цепочки u считается верхним символом магазина; если u = e, то магазин считается пустым.

Расширенный МП-автомат M = (Q, T, Г, D, q_0, Z_0, F) называется детерминированным, если выполнены следующие условия:

(1) Множество D(q, a, u) содержит не более одного элемента для любых $q \in Q, a \in T \cup \{e\}, u \in$ Г*,

(2) Если D(q, a, u), D(q, a, v) и u v, то не существует цепочки x такой, что u=vx или v=ux,

(3) Если D(q, a, u), D(q, e, v), то не существует цепочки x такой, что u = vx или v = ux.

В названии LR(1) символ L указывает на то, что входная цепочка читается слева-направо, R – на то, что строится правый вывод, наконец, 1 указывает на то, что анализатор видит один символ непрочитанной части входной цепочки.

LR(1)-анализ привлекателен по нескольким причинам:

- LR(1)-анализ - наиболее мощный метод анализа без возвратов типа сдвиг-свертка;
- LR(1)-анализ может быть реализован довольно эффективно;
- LR(1)-анализаторы могут быть построены для практически всех конструкций языков программирования;
- класс грамматик, которые могут быть проанализированы LR(1)-методом, строго включает класс грамматик, которые могут быть проанализированы предсказывающими анализаторами (сверху-вниз типа LL(1)).

Анализатор состоит из входной ленты, выходной ленты, магазина, управляющей программы и таблицы анализа (LR(1)-таблицы), которая имеет две части – функцию действий (Action) и функцию переходов (Goto). Управляющая программа одна и та же для всех LR(1)-анализаторов, разные анализаторы отличаются только таблицами анализа.

Анализатор читает символы на входной ленте по одному за шаг. В процессе анализа используется магазин, в котором хранятся строки вида $S_0X_1S_1X_2S_2$ ... X_mS_m (S_m – верхушка магазина). Каждый X_i – символ грамматики (терминальный или нетерминальный), а S_i – символ состояния. Заметим,

что символы грамматики (либо символы состояний) не обязательно должны размещаться в магазине. Однако, их использование облегчает понимание поведения LR-анализатора.
Элемент функции действий Action[S_m; a_i] для символа состояния S_m и входа $a_i \in T \cup \{\}$ может иметь одно из четырех значений:

- shift S (сдвиг), где S - символ состояния,
- reduce A -> γ (свертка по правилу грамматики A -> γ),
- accept (допуск),
- error (ошибка).

Элемент функции переходов Goto[S_m; A] для символа состояния S_mи входа может иметь двух значений:
S, где S - символ состояния,
error (ошибка).
Конфигурацией LR(1)-анализатора называется пара, первая компонента которой – содержимое магазина, а вторая – непросмотренный вход:
($S_0X_1S_1X_2S_2$ ... X_mS_m, a_ia_{i+1} ... $a_n$$)
Основа сентенциальной формы.
Префиксы правых сентенциальных форм, которые могут появиться в магазине анализатора, называются активными префиксами. Основа сентенциальной формы всегда располагается на верхушке магазина. Таким образом, активный префикс – это такой префикс правой сентенциальной формы, который не переходит правую границу основы этой формы.
Анализатор может проделать один из следующих шагов:

1) Если Action[S_m, a_i] = shift S, то анализатор выполняет сдвиг, переходя в конфигурацию
$$(S_0X_1S_1X_2S_2 \dots X_mS_ma_iS, a_{i+1} \dots a_n\$)$$
То есть, в магазин помещаются входной символ a_i и символ состояния S, определяемый Action[S_m, a_i]. Те- кущим входным символом становится a_{i+1}.
2) Если Action[S_m, a_i] = reduce A -> γ, то анализатор выполняет свертку, переходя в конфигурацию
$$(S_0X_1S_1X_2S_2 \dots X_{m-r}S_{m-r}AS, a_ia_{i+1} \dots a_n\$)$$
где S = Goto[S_{m-r}; A] и r - длина γ, правой части правила вывода. Анализатор сначала удаляет из магазина 2r символов (r символов состояния и r символов грамматики), так что на верхушке оказывается состояние S_{m-r}. Затем анализатор помещает в магазин A - левую часть правила вывода, и S - символ состояния, определяемый Goto[S_{m-r}, A]. На шаге свертки текущий входной символ не меняется. Для LR(1)-анализаторов последовательность символов грамматики X_{m-r+1} ... X_m, удаляемых из магазина , всегда соответствует γ - правой части правила вывода, по которому делается свертка. После осуществления шага свертки генерируется выход LR(1)-анализатора, то есть исполняются семантические действия, связанные с правилом, по которому делается свертка, например, печатаются номера правил, по которым делается свертка. Заметим, что функция Goto таблицы анализа, построенная по грамматике G, фактически представляет собой функцию переходов детерминированного конечного автомата, распознающего активные префиксы G.
3) Если Action[S_m, a_i] = accept, то разбор успешно завершен.
4) Если Action[S_m, a_i] = error, то анализатор обнаружил ошибку, и выполняются действия по диагностике и восстановлению.

LR(1) ситуация.
LR(1)-ситуацией называется пара [A->(?).β, a], где A->β – правило грамматики, a – терминал или правый концевой маркер $. Вторая компонента ситуации называется аванцепочкой.
Будем говорить, что LR(1)-ситуация [A .β, a] допустима для активного префикса +, если существует вывод $S \Rightarrow_\tau^* \gamma Aw \Rightarrow_\tau \gamma\alpha\beta w$, где + = γ и либо a – первый символ w, либо w = e и a = $.
Конструирование канонической системы множеств допустимых LR(1)-ситуаций.
Вход. КС-грамматика G = (N, T, P, S).
Выход. Каноническая система C множеств допустимых LR(1)-ситуаций для грамматики G.
Метод. Выполнить для пополненной грамматики G' процедуру items, которая использует функции closure и goto.

```
function closure(I){ /* I – множество ситуаций */
 do{
for (каждой ситуации [A .Bβ, a] из I,
каждого правила вывода B γ из G', каждого терминала b из FIRST(βa),
такого, что [B .γ, b] нет в I)
добавить [B .γ, b] к I; }
 while (к I можно добавить новую ситуацию);
return I; }
```

```
function goto(I,X){ /* I – множество ситуаций;
                    X – символ грамматики */
ПустьJ={[A X.β;a]|[A .Xβ,a]\inI};
  return closure(J);
 }
procedure items(G'){ /* G' – пополненная
                грамматика */
I' = closure({[S' .S, $]}); C = {I0}; do{
    for (каждого множества ситуаций I из
     системы C, каждого символа грамматики X
      такого, что goto(I, X) не пусто
     и не принадлежит C)
     добавить goto(I, X) к системе C;
   }
   while (к C можно добавить новое множество
ситуаций);
```

Вход. Каноническая система C = {I_0, I_1, ... , I_n} множеств допустимых LR(1)-ситуаций для грамматики G.

Выход. Функции Action и Goto, составляющие LR(1)- таблицу для грамматики G.

Метод. Для каждого состояния i функции Action[i, a] и Goto[i, X] строятся по множеству ситуаций I_i:

- Значения функции действия (Action) для состояния i определяются следующим образом:
  - Если $[A \to \alpha.a\beta, b] \in I_i$ (a - терминал) и goto(I_i, a)= I_j , то полагаем Action[i, a] = shift j;
  - Если $[A \to \alpha;., a] \in I_i$, причем A->S', то полагаем Action[i, a] = reduce A ;
  - Если $[S' \to S., \$] \in I_i$, то полагаем Action[i, $] = accept.
- Значения функции переходов для состояния i определяются следующим образом: если goto(I_i, A) = I_j , то Goto[i, A] = j (здесь A - нетерминал).
- Все входы в Action и Goto, не определенные шагами 2 и 3, полагаем равными error.
- Начальное состояние анализатора строится из множества, содержащего ситуацию [S' .S, $].

Таблица на основе функций Action и Goto, полученных в результате работы алгоритма, называется канонической LR(1)-таблицей. Работающий с ней LR(1)-анализатор, называется каноническим LR(1)- анализатором.

Если грамматика не является LR(1), то анализатор типа сдвиг-свертка при анализе некоторой цепочки может достигнуть конфигурации, в которой он, зная содержимое магазина и следующий входной символ, не может решить, делать ли сдвиг или свертку (конфликт сдвиг/свертка), или не может решить, какую из нескольких сверток применить (конфликт свертка/свертка).

14. Сложность алгоритма как функция одного или нескольких числовых аргументов. Сложность в худшем случае.

Определение . Пусть на возможных входах x алгоритма A определена неотрицательная числовая функция $\|x\|$ (размер входа). Пусть также определены целочисленные неотрицательные функции $C_A{}^T(x)$, $C_A{}^S(x)$ временных и пространственных затрат алгоритма A. Тогда *временной и пространственной сложностями A* называются функции числового аргумента

$$T_A(n) = \max_{\|x\|=n} C_A^T(x), S_A(n) = \max_{\|x\|=n} C_A^S(x)$$

(областью изменения n как аргумента функций $T_A(n)$ и $S_A(n)$ является множество значений размера $\|\cdot\|$). Более полно каждая такая сложность именуется сложностью *в худшем случае*.

Пусть функция $C_A{}^T(x)$ в определении отражает затраты на выполнение лишь какого-то одного типа операций в предположении, что все эти операции требуют одинакового времени выполнения, не зависящего от вида и размера операндов, и это время равно 1. Тогда соответствующая функция $T_A(n)$ — это сложность *по числу операций* рассматриваемого типа. Привлечение в качестве $C_A{}^S(x)$ функции, отражающей только количество хранимых дополнительных величин того или иного фиксированного типа, приводит к пространственной сложности $S_A(n)$ *по числу величин* рассматриваемого типа. Оно же – алгебраическая сложность.

Алгебраическая сложность по умножениям и делениям – мультипликативная сложность.

Определение . Функции $f(n)$ и $g(n)$ имеют *одинаковый порядок* (пишут $f(n) = \Theta(g(n))$) тогда и только тогда, когда найдутся положительные c_1, c_2, N такие, что неравенства $c_1|g(n)| \le |f(n)| \le c_2|g(n)|$ выполнены для всех $n > N$.

Соотношение $f(n) = \Omega(g(n))$ имеет место тогда и только тогда, когда найдутся положительные c,N такие, что для всех $n>N$ выполнено $|f(n)| \ge c|g(n)|$.

Определение . Если имеет место оценка $f(n) = O(g(n))$, то она называется *точной*, коль скоро существует неограниченно возрастающая последовательность неотрицательных целых чисел $\{n_k\}$ такая, что для $\varphi(k) = f(n_k)$, $\psi(k) = g(n_k)$ имеет место $\varphi(k) = \Omega(\psi(k))$.

Далее $T_A^*(m) = \max_{2^{m-1} \le n < 2^n} T_A(n)$

Лемма 1. Пусть f(x)—неубывающая функция вещественной переменной. Тогда если $T_A{}^*(m) \le f(m)$, то $T_A(n) \le f(\log_2 n + 1)$.

Доказательство. Пусть m и n фиксированы, $2^{m-1} \le n < 2^m$, и пусть значение $\hat{n}$ таково, что

$$2^{m-1} \le \hat{n} < 2^m$$

и при этом

$$T_A(\hat{n}) = T_A^*(m)$$

Используем неубывание f(x):

$$T_A(n) \le T_A(\hat{n}) = T_A^*(m) \le f(m) = f(\lfloor \log_2 n \rfloor + 1 \le f(\log_2 n + 1)$$

Лемма . . Пусть g(x)—неубывающая функция вещественной переменной. Тогда

(i) если $T_A(n) \le g(n)$, то $T_A{}^*(m) \le g(2^m)$,

(ii) если $T_A(n) \ge g(n)$, то $T_A{}^*(m) \ge g(2^{m-1})$.

Доказательство. Пусть m и n фиксированы, $2^{m-1} \le n < 2^m$, и пусть значение $\hat{n}$ таково, что

$$2^{m-1} \le \hat{n} < 2^m$$

и при этом

$$T_A(\hat{n}) = T_A^*(m)$$

Используем неубывание g(x):

(i) $T_A^*(m) = T_A(\hat{n}) \le g(\hat{n}) \le g(2^m)$

(ii) $T_A^*(m) = T_A(\hat{n}) \ge g(\hat{n}) \ge g(2^{m-1})$

Из Лемм напрямую

Теорема . . Пусть f(x)—неубывающая функция вещественной переменной. Тогда если $T_A{}^*(m)=O(f(m))$, то $T_A(n)=O(f(\log_2 n+1))$; как следствие, при $f(x + 1) = O(f(x))$ имеем $T_A(n) = O(f(\log_2 n))$.

Теорема . . Пусть g(x)—неубывающая функция вещественной переменной. Тогда

(i) если $T_A(n) = O(g(n))$, то $T_A{}^*(m) = O(g(2^m))$

(ii) если $T_A(n)=\Omega(g(n))$, то $T_A^*(m)=\Omega(g(2^{m-1}))$; как следствие, при $g\left(\frac{x}{2}\right)=\Omega(g(x))$ имеем $T_A^*(m) = \Omega(g(2^m))$.

15. Сложность в среднем. Сложность рандомизированного алгоритма.

Определение 1**.** Пусть при любом допустимом значении *s* множество X_S всех входов размера *s* является вероятностным пространством, в силу чего временные и пространственные затраты алгоритма *A* для входов *x* (т. е. $C_A{}^T(x)$ и $C_A{}^S(x)$) размера *s* являются случайными величинами на X_S.

Сложностью в среднем называется математическое ожидание соответствующей случайной величины.

$$\sum_{x \in X_s} C_A^T(x) P_s(x) \text{ и } \sum_{x \in X_s} C_A^S(x) P_S(x)$$

Теорема . Для любого алгоритма A при любом распределении вероятностей на множестве X_S, где s — некоторое допустимое значение размера $\|\cdot\|$, сложность в среднем не превосходит сложности в худшем случае.

Доказательство:

$$\bar{T}_A(S) = \sum_{x \in X_s} C_A^T(x) P_s(x) \leq \sum_{x \in X_s} \left(\max_{x \in X_s} C_A^T(x) \right) P_s(x) = \sum_{x \in X_s} T_A(x) P_s(x) = T_A(s)$$

Аналогично второе

Определение . Вещественная неотрицательная функция *f*(*m*), определенная для целых положительных значений аргумента, называется *полиномиально ограниченной*, если существует полином *P*(*m*) с вещественными коэффициентами такой, что $f(m) \leq P(m)$ для всех $m \in \mathrm{N}^+$. (Очевидно, что мы получим эквивалентное определение, если дополнительно потребуем, чтобы все коэффициенты полинома *P*(*m*) были неотрицательными; другие эквивалентные варианты определения: $f(m) = O(m^d)$ для некоторого $d \in \mathrm{N}$ и $f(m) = m^{O(1)}$.)

Определение 3**.** Алгоритмы с элементами случайности, реализуемыми обращениями к генераторам случайных чисел, называются *рандомизированными*.

Принцип Яо: Пусть для размера входа зафиксировано некоторое значение n, и пусть все входы размера n образуют конечное множество X, на котором задано некоторое распределение вероятностей. Пусть $\mathcal{A}$ — конечное множество алгоритмов, применимых к элементам множества X, и пусть на $\mathcal{A}$ тоже задано некоторое распределение вероятностей. Множество $\mathcal{A}$ с этим распределением можно рассматривать как единый рандомизированный алгоритм, считая его сложностью T(n) усредненные затраты в худшем случае. Тогда, если найти некоторую нижнюю границу f(n) сложностей в среднем (в соответствии с данным распределением вероятностей на множестве X входов размера n) всех детерминированных алгоритмов класса $\mathcal{A}$, то, независимо от конкретного вида распределений на X и $\mathcal{A}$, будет выполняться неравенство $f(n) \leq T(n)$.

16. Основные принципы построения сети Интернет. Иерархическая модель компьютерной сети. Адресация в сети Интернет, протоколы ARP, DHCP. Модели основных протоколов IP, TCP, ICMP. Модель взаимодействия приложений в Интернет.

Модель OSI имеет уровневую организацию. Она включает в себя **семь уровней**: физический, канальный, сетевой, транспортный, сессии, представления и прикладной.

Модель TCP/IP включает в себя **3 основных уровня**:

- межсетевой уровень
- транспортный уровень
- уровень приложений.

В модели TCP/IP нет уровней сессии и представления, поскольку необходимость в них была неочевидна для ее создателей.

Internet – метасеть, состоящая из многих сетей, которые работают согласно протоколам семейства TCP/IP, объединены через шлюзы и используют единое адресное пространство и пространство имен.

Среду передачи данных в Internet нельзя рассматривать только как паутину проводов или оптоволоконных линий. Оцифрованные данные пересылаются через **маршрутизаторы**, которые соединяют сети и с помощью сложных алгоритмов выбирают наилучшие маршруты для информационных потоков.

Глобальная сеть включает подсеть связи, к которой подключаются локальные сети, отдельные компоненты и терминалы (средства ввода и отображения информации).

Компьютеры, за которыми работают пользователи-клиенты, называются **рабочими станциями**, а компьютеры, являющиеся источниками ресурсов сети, предоставляемых пользователям, называются **серверами**. Такая структура сети получила название **узловой**.

Инфраструктура Интернет:

- Магистральный уровень (система связанных высокоскоростных телекоммуникационных серверов).
- Уровень сетей и точек доступа (крупные телекоммуникационные сети), подключенных к магистрали.
- Уровень региональных и других сетей.
- ISP – интернет-провайдеры.
- Пользователи.

Адресация.

Есть две таблицы. Первая показывает как достичь интересующей сети. Вторая – как достичь узел внутри сети. Когда поступает IP пакет, маршрутизатор ищет его адрес доставки в таблице маршрутизации. Если это адрес другой сети, то пакет передают дальше тому маршрутизатору, который отвечает за связь с этой сетью. Если это адрес в локальной сети, то маршрутизатор направляет пакет прямо по месту назначения. Если адреса нет в таблице, то маршрутизатор направляет пакет специально выделенному по умолчанию маршрутизатору, который должен разобраться с этим случаем с помощью более подробной таблицы. Такая организация алгоритма позволяет существенно сократить размер таблиц в маршрутизаторах. С появлением подсети структура адресов меняется. Теперь записи в таблице имеют форму (эта_сеть, подсеть, 0) и (эта_сеть, эта_подсеть, машина). Таким образом, маршрутизатор подсети в данной локальной сети знает, как достичь любую подсеть в данной локальной сети, и как найти конкретную машину в своей подсети. Все что ему нужно – это знать маску подсети. С помощью логической операции И маршрутизатор выделяет адрес подсети с помощью маски. По своим таблицам он определяет как достичь нужной подсети или, если этого локальная подсеть данного маршрутизатора, как достичь конкретной машины.

Address Resolution Protocol – протокол определения адреса.

Ethernet адрес. Этот адрес имеет 48 разрядов. Сетевая карта знает только такие адреса и ничего об 32-разрядных IP. Как отобразить 32-разрядный IP адрес на адреса канального уровня, например, Ethernet адрес. Для отображения IP адреса на Ethernet адрес, в подсеть посылается запрос у кого такой IP адрес. Машина с указанным адресом шлет ответ. Протокол, который реализует рассылку запросов и сбор ответов – ARP протокол. Практически каждая машина в Internet имеет этот протокол. Для того чтобы узнать адрес в другой сети есть два решения – есть определенный маршрутизатор, который принимает все сообщения, адресованные определенной сети или группе адресов – proxy ARP. Этот маршрутизатор знает как найти адресуемую машину. Другое решение – выделенный маршрутизатор, который управляет маршрутизацией удаленного трафика. Машина определяет, что обращение идет в удаленную сеть и шлет сообщение на этот маршрутизатор.

DHCP (англ. Dynamic Host Configuration Protocol – протокол динамической настройки узла) – сетевой протокол, позволяющий компьютерам автоматически получать IP-адрес и другие параметры, необходимые для работы в сети TCP/IP. Данный протокол работает по модели «клиент-сервер». Для автоматической конфигурации компьютер-клиент на этапе конфигурации сетевого устройства обращается к так называемому серверу DHCP, и получает от него нужные параметры. Сетевой администратор может задать диапазон адресов, распределяемых сервером среди

компьютеров. Это позволяет избежать ручной настройки компьютеров сети и уменьшает количество ошибок. Протокол DHCP используется в большинстве сетей TCP/IP.
Протокол DHCP предоставляет три способа распределения IP-адресов:

- **Ручное распределение.** При этом способе сетевой администратор сопоставляет аппаратному адресу (для Ethernet сетей это MAC-адрес) каждого клиентского компьютера определённый IP-адрес. Фактически, данный способ распределения адресов отличается от ручной настройки каждого компьютера лишь тем, что сведения об адресах хранятся централизованно (на сервере DHCP), и потому их проще изменять при необходимости.
- **Автоматическое распределение.** При данном способе каждому компьютеру на постоянное использование выделяется произвольный свободный IP-адрес из определённого администратором диапазона.
- **Динамическое распределение.** Этот способ аналогичен автоматическому распределению, за исключением того, что адрес выдаётся компьютеру не на постоянное пользование, а на определённый срок. Это называется арендой адреса. По истечении срока аренды IP-адрес вновь считается свободным, и клиент обязан запросить новый (он, впрочем, может оказаться тем же самым). Кроме того, клиент сам может отказаться от полученного адреса.
  Некоторые реализации службы DHCP способны автоматически обновлять записи DNS, соответствующие клиентским компьютерам, при выделении им новых адресов. Это производится при помощи протокола обновления DNS, описанного в RFC 2136. Протокол DHCP является клиент-серверным, то есть в его работе участвуют клиент DHCP и сервер DHCP. Передача данных производится при помощи протокола UDP, при этом сервер принимает сообщения от клиентов на порт 67 и отправляет сообщения клиентам на порт 68.

Модель сервиса IP

Internet Protocol («межсетевой протокол») – маршрутизируемый протокол сетевого уровня стека TCP/IP. Именно IP стал тем протоколом, который объединил отдельные компьютерные сети во всемирную сеть Интернет. Неотъемлемой частью протокола является адресация сети.
IP объединяет сегменты сети в единую сеть, обеспечивая доставку пакетов данных между любыми узлами сети через произвольное число промежуточных узлов (маршрутизаторов). Он классифицируется как протокол третьего уровня по сетевой модели OSI. IP не гарантирует надёжной доставки пакета до адресата – в частности, пакеты могут прийти не в том порядке, в котором были отправлены, продублироваться (приходят две копии одного пакета), оказаться повреждёнными (обычно повреждённые пакеты уничтожаются) или не прийти вовсе. Гарантию безошибочной доставки пакетов дают некоторые протоколы более высокого уровня – транспортного уровня сетевой модели OSI, – например, TCP, которые используют IP в качестве транспорта.
В современной сети Интернет используется IP четвёртой версии, также известный как IPv4. В протоколе IP этой версии каждому узлу сети ставится в соответствие IP-адрес длиной 4 октета (4 байта). При этом компьютеры в подсетях объединяются общими начальными битами адреса. Количество этих бит, общее для данной подсети, называется маской подсети (ранее использовалось деление пространства адресов по классам – A, B, C; класс сети определялся диапазоном значений старшего октета и определял число адресуемых узлов в данной сети, сейчас используется бесклассовая адресация).

- предотвращает «зацикливание» пакетов;
- фрагментирует пакеты если они слишком длинные;
- использует контрольную сумму, чтобы сократить возможность доставки в неправильное место назначение;
- две версии: IPv4 с 32 битным адресом IPv6 с 128 битным адресом
- позволяет добавлять новые опции к заголовку;
- работает над любой физической средой;
- над IP работают различные транспортные протоколы;
- поверх транспорта работают прикладные протоколы;
- IP используется всегда для передачи пакетов между различными сетями;
- очень простой сервис.

Модель сервиса TCP

Доступ к TCP-сервису происходит через сокет. Сокет состоит из IP-адреса хоста и 16-разрядного локального номера на хосте, называемого порт. Сокеты создаются как отправителем, так и получателем. Порт — это TSAP для TCP. Каждое соединение идентифицируется парой сокетов, между которыми оно установлено. Один и тот же сокет может быть использован для разных соединений. Никаких дополнительных виртуальных соединений не создается.
Порты до 256 номера зарезервированы для стандартных сервисов, которые постоянно активны и готовы к работе. Например, для обеспечения FTP-передачи файла соединение должно выполняться через 21-й порт, где находится FTP-демон, а для TELNET – через 23-й порт. Полный список таких портов можно найти в RFC 1700.

Все TCP-соединепня — дуплексные, т.е. передача по ним происходит независимо в оба направления. TCP-соединение поддерживает только соединение точка—точка. Не существует TCP-соединений типа «от одного ко многим».
TCP-агент поддерживает поток байтов, а не поток сообщений. Напомним, это означает, что гранины сообщений не поддерживаются автоматически в потоке. Например, если по TCP-соединению передается текст, разбитый на страницы, то TCP-агент не будет каким-либо образом обозначать конец каждой страницы.
После передачи приложением данных TCP-агенту, эти данные могут быть отправлены сразу на сетевой уровень, а могут быть буферизованы. Как поступить в этом случае решает TCP-агент. Однако в ряде случаев бывает необходимо, чтобы данные были отправлены сразу, например если эти данные представляют собой команду для удаленной машины. Для этого в заголовке TCP-сегмента имеется флаг PUSH, и если он установлен, то это говорит TCP-агенту о том, что данные должны быть переданы немедленно.
Наконец, если в заголовке TPDU-сегмента установлен флаг URGENT, то TCP-агент передает такой сегмент незамедлительно. Когда срочные данные поступают к месту назначения, то их передают получателю немедленно.

Модель сервиса ICMP

Управление функционированием Интернета на сетевом уровне происходит через маршрутизаторы с помощью протокола ICMP (Internet Control Message Protocol), описанного в RFC 792. Этот протокол обеспечивает доставку сообщений любой машине, имеющей IP-адрес, от маршрутизаторов и других хостов в сети. С помощью этого протокола реализуют обратную связь для решения проблем, возникающих при передаче. Он также выявляет и рассылает сообщения о десятках событий. Для доставки своих сообщений протокол ICMP использует пакеты IP-протокола. Приведем наиболее важные сообщения.
Сообщение **destination unreachable** охватывает множество случаев, например, случай, когда маршрутизатор не знает, как достигнуть необходимой подсети или хоста, или случай, когда дейтаграмма при доставке должна быть фрагментирована, но установлен флаг, который запрещает это делать.
Сообщение **time exceeded** посылает маршрутизатор, если он обнаружил дейтаграмму с истекшим времени жизни. Это сообщение также генерирует хост, если он не успел завершить обработку IP-пакета до истечения времени его жизни. (Далее слова «пакет», «IP-пакет», «дейтаграмма» будем понимать как синонимы, если специально не оговорено что-либо другое.)
Синтаксические или семантические ошибки в заголовке IP-пакета вызывают появление сообщения **parameter problem**.
Сообщение **source quench** обеспечивает управление потоком. Маршрутизатор или хост-получатель высылает этот пакет хосту-отправителю, если последнему необходимо понизить скорость передачи. Другими словами, это пример подавляющего пакета. Сообщения такого типа будут генерироваться до тех пор, пока скорость поступления пакетов от отправителя не достигнет значения, необходимого хосту- получателю. Это сообщение система может использовать для предотвращения перегрузки, поскольку оно возникает всякий раз, когда маршрутизатор вынужден сбросить пакет из-за переполнения своего буфера.
Сообщение **redirect** позволяет маршрутизатору отправить рекомендацию о лучшем маршруте и впредь посылать пакеты с определенным IP-адресом через другой маршрутизатор.
Сообщения **echo request** и **echo reply** позволяют проверить работоспособность хостов в сети: получатель сообщения echo request обязан ответить сообщением echo reply, причем с теми же параметрами, что и в echo request.
Сообщения **time-stamp request** и **time-stamp reply** позволяют измерять временную задержку в Интернете на сетевом уровне. Этот механизм необходим, например, для работы алгоритма маршрутизации по состоянию канала.

Свойство	Поведение
Сообщение о состоянии	Самодостаточное сообщение об ошибке или исключительном состоянии
Ненадежный	Сервис без соединения и подтверждения

ICMP предоставляет информацию о сетевом уровне хостам и маршрутизаторам. ICMP работает над IP и относится к транспортному уровню. ping и traceroute реализованы с помощью ICMP.

17. Физический уровень стека сетевых протоколов. Технологии Ethernet и WiFi. Алгоритмы работы, коллизии, управление множественным доступом к каналу.

Назначение физического уровня – передавать данные в виде потока битов от одной машины к другой. При этом для передачи данных можно использовать различные физические среды, каждую из которых характеризует следующие параметры:

- Ширина полосы пропускания.
- Пропускная способность.
- Задержка сигнала.
- Стоимость.
- Сложность прокладки.
- Сложность прокладки.
- Сложность обслуживания.
- Достоверность передачи.
- Затухание.
- Помехоустойчивость.

Кабели в стандарте IEEE 802.3.

Первым появился так называемый толстый Ethernet(10Base5). Коаксиальный кабель жёлтого цвета с отметками через каждые 2.5 м, указывающими, где можно производить подключение. Подключение выполняется через специальные розетки, которые монтируются прямо на кабеле. В эти розетки встраивался специальный прибор – трансивер, отвечающий за обнаружение несущей частоты и коллизий. Когда трансивер обнаруживает коллизию, он посылает специальный сигнал по кабелю, гарантирующий, что другие трансиверы услышат эту коллизию. Кабель обеспечивает пропускную способность 10 Мбит/с, а максимальная длина равна 500м(10Base5).

Вторым появился 10Base2. Это более простой в употреблении коаксиальный кабель с простым подключением через BNC-коннектор, представляющий собой Т-образное соединение коаксиальных кабелей. Его сегмент не должен превышать 200м и объединять более 30 машин.

Решение проблемы поиска обрыва, частичного повреждения кабеля или плохого контакта в коннекторе привели к созданию совершенно иной кабельной конфигурации на основе витой пары. В этом случае каждая машина соединяется витой парой со специальным устройством – хабом. Такой способ обозначается 10Base-T.

Трансиверы в 10Base5 размещаются прямо на кабеле и соединяются с компьютером трансиверным кабелем, длина которого не может превышать 50м. Трансиверный кабель состоит из пяти витых пар. Две из них используются для передчи данных к компьютеру и от него, две служат для передачи управляющей информации в обе стороны, а пятая – для подачи питания на трансивер.

Трансиверный кабель подключается к контроллеру в компьютере через интерфейс AUI. В контроллере имеется специальная микросхема, отвечающая за приём кадром и их отправку, проверку и формирование контрольной суммы. В некоторых случаях эта микросхема отвечает и за управление буферами на канальном уровне, очередью буферов на отправку и обеспечивает прямой доступ к памяти машины, а также решает другие вопросы доступа к сети.

В 10Base2 трансивер располагается на контроллере, и каждая машина должна иметь свой индивидуальный трансивер.

В 10Base-T трансивера нет вовсе. Машины соединяются хабом с витой парой, длина которой не должна превышать 100м. Вся электроника сосредоточена в нём.

Последний используемый в стандарте IEEE 802.3 тип кабеля – оптоволокно 10Base-F, которое относительно дорогое, но обеспечение низкого уровня шума и большая длина одного сегмента являются его преимуществом. Для увеличения длины сегментов в этом стандарте используются репитеры – устройства физического уровня, отвечающие за очистку, усиление и передачу сигнала. Репитеры не могут отстоять друг от друга более чем на 2.5 км, и на одном сегменте их не может быть более четырёх.

Физический уровень. Первой и ключевой технологией стандарта 802.11 является технология расширения спектра передачи методом **прямой последовательности** (Direct Sequence Spread Spectrum – DSSS). Использование DSSS позволяет беспроводным интерфейсам передавать данные со скоростью от 1 до 2 Мбит/с.

Идея метода. Пусть имеется канал с широкой полосой пропускания. Разобьем его на полосы. Каждому значению бита сопоставим определенный кот с длиной, равной числу полос, на которые разбили канал. Теперь будем передавать каждый бит, параллельно передавая его код, причем каждый элемент кода(чип) в своей полосе. Такой способ передачи позволяет эффективнее использовать полосу пропускания канала, и он более надежен по сравнению с традиционным способом передачи.

Канальный уровень. Минимально сеть wifi может содержать всего два устройства. В этом случае организуется выделенная есть, в которую входят все беспроводные интерфейсы этих устройств. Данное соединение можно сравнить с соединением типа точка-точка в проводной связи.

Для решения задачи совместимости в сети устанавливается выделенный узел – точка доступа, представляющая собой устройство, имеющее проводной интерефейс для подключения к проводной

сети, а также антенну, образующую вокруг себя зону покрытия точки доступа. Радиус покрытия точки доступа от 90 до 150м.
Когда абонентское устройство включается внутри сети, оно начинает прослушивать эфир в поисках совместимого устройства, с которым оно могло бы взаимодействовать. Этот этап называется **сканированием**. При активном сканировании генерируется широковещательный запрос от абонентского устройства, обязательно включающий в себя идентификатор сети (SSID), к которой он хочет присоединиться. Когда запрос достигает точки доступа, имеющий запрашиваемый идентификатор, эта сеть генерирует ответ на запрос. При **пассивном сканировании** абонентское устройство слушает эфир и ожидает появления кадров-маяков, которые периодически рассылаются точками доступа. Когда абонентское устройство получает кадр-маяк, в котором указан SSID, оно пытается присоединиться к указанной сети. Пассивное сканирование – постоянный процесс, при котором устройства могут присоединиться к точке доступа или отсоединяться по мере изменения мощности радиосигнала.
Структура кадра. В wifi определены 3 типа кадров: контрольные, управляющие и кадры данных. Структрура совпадает с IEEE802.3, за исключением некоторых отличий: размер поля данных равен 1500 байт, максимальный размер кадра составляет 2346 байт.

Базовая модель динамического предоставления доступа к каналу.

Пять основных предположений, составляющих основу моделей сетей ЭВМ, в которых в качестве СПД используется канал с множественным доступом:

1) Станции. Модель состоит из N независимых станций (компьютеров, телефонов, факс-машин и т. п.). На каждой станции работает пользователь или программа, генерирующие кадры для передачи. Предполагаем, что если кадр сгенерирован, то станция блокируется, и новый кадр не появится, пока не будет передан первый кадр. Это означает, что станции независимы, и на каждой из них работает только одна программа или один пользователь, генерирующие нагрузку с постоянной скоростью.
2) Единственность канала. Канал один и он доступен всем станциям. Все станции равноправны. Они получают кадры и передают кадры только через этот единственный канал. Аппаратные средства всех станций для доступа к каналу одинаковые, но программно можно устанавливать станциям приоритеты.
3) Коллизии. Если во время передачи кадра одной станцией другая станция начала передачу своего кадра, то такой случай будем называть коллизией. Предполагаем, что любая станция может обнаружить коллизию и что кадры, разрушенные при коллизии, должны быть посланы повторно позднее. Кроме коллизий других ошибок передачи нет.
4) Время. Возможны две модели времени – непрерывная и дискретная:
    a. непрерывное время. Передача кадра может начаться в любой момент. В сети нет единых часов, которые разбивают время на слоты. Другими словами, время является непрерывной функцией, отображающей интересующие нас действия в сети на множество вещественных чисел;
    b. дискретное время. Все время работы канала разбивается на одинаковые интервалы, называемые слотами, В слоте может оказаться нуль кадров, если это слот ожидания, один кадр, если в этом слоте передача кадра прошла успешно, и несколько кадров, если в этом слоте произошла коллизия.
5) Доступ к каналу. Возможны два способа доступа станции к каналу:
    a. с обнаружением несущей. Станция прежде чем использовать канал всегда определяет, занят он или нет с помощью несущей – сигнала определенной формы. Когда канал не занят, по нему все время передается такой сигнал, а если канал занят, то сигнал в нем отличается от несущей, и станция не начинает передачу;
    b. при отсутствии несущей. Станция ничего не знает о состоянии канала, пока не начнёт использовать его. Она сразу начинает передачу и лишь в ходе передачи обнаруживает коллизию, т.к. сигнал, который она «увидит» в канале, будет отличаться от того сигнала, который станция передала в канал.

Говоря о **динамическом доступе**, подразумевают, что отсутствует какая-либо фиксированная политика предоставления доступа к каналу для передачи в отличие от статических методов доступа. При этом любая станция может запросить доступ к каналу в любой момент времени, а методы, доступа лишь определяют правила удовлетворения этих запросов.

Методы множественного доступа ALOHA. Система состояла из наземных радиостанций, работающих на одной частоте и связывающих острова между собой. Идея ее конструкции заключалась в том, чтобы позволить в вещательной среде любому количеству пользователей неконтролируемо использовать один и тот же канал.

Чистая ALOHA: любой пользователь, желающий передать сообщение, сразу пытается это сделать. Благодаря тому, что в вещательной среде у него всегда есть обратная связь, т.е. он может определить, пытался ли кто-то еще передавать сообщение на его частоте, отправитель может установить возникновение конфликта при передаче. Обратная связь в среде ЛВС происходит практически мгновенно. Отправитель при этом должен слушать среду передачи до тех пор, пока последний бит его сообщения не достигнет самого отдаленного получателя. Обнаружив конфликт, отправитель ожидает некоторый случайный отрезок времени, после чего повторяет попытку

передачи. Интервал времени на ожидание должен быть случайным, иначе конкуренты, повторяя попытки передачи вызовут коллизию снова. Системы, в которых пользователи конкурируют за получение доступа к общему каналу, называются **системами с состязаниями.**
Неважно, когда произошел конфликт, оба кадра считаются испорченными и должны быть переданы повторно. Контрольная сумма, защищающая данные в кадре, не позволяет различать разные случаи наложения кадров.
Назовем **временем кадра** время, необходимое на передачу кадра стандартной фиксированной длины. Обозначим это время t. Предположим, что число пользователей неограниченно и все они порождают кадры по закону Пуассона со средним числом N кадров за t. Это означает, что вероятность события, при котором будет порождено n кадров за время t, можно записать в виде:

$$P[k] = \frac{G^k e^{-G}}{k!}$$

$$P[n] = \frac{\lambda^n e^{-\lambda}}{n!}, \lambda = N$$

Поскольку при N > 1 очередь на передачу будет только расти и все кадры будут страдать от коллизий, предположим, что 0 < N < 1. Также предположим, что вероятность за время кадра сделать к попыток передачи, как новых, так и ранее не переданных из-за коллизий кадров, распределяется по закону Пуассона со средним значением G. Понятно, что при этом должно выполняться соотношение □ ≥ □, иначе очередь будет расти бесконечно. При слабой загрузке (□ ∼ 0) будет мало передач, а, следовательно, и коллизий, поэтому G N. При высокой загрузке должно выполняться соотношение G > N. При этом пропускная способность канала (S) будет равна числу кадров, которые надо передать, умноженному на вероятность успешной передачи. Если обозначить P0 вероятность отсутствия коллизий при передаче кадра, то можно записать □ = □□$_0$. Рассмотрим внимательно, сколько времени требуется отправителю, чтобы обнаружить коллизию. Пусть он начал передачу в момент времени □$_0$ и пусть требуется время t, чтобы кадр достиг самой отдаленной станции. Тогда, если в тот момент, когда кадр почти достиг этой отдаленной станции, она начнет передачу (ведь в системе ALOHA. станция сначала передает, а потом слушает), отправитель узнает об этом только через время, равное □$_0$ + 2□. Вероятность появления к кадров при передаче кадра с распределением Пуассона поэтому вероятность, что появится 0 кадров, равна $-□$. За двойное время кадра среднее число кадров равно 2G, откуда □$_0$ = □$^{-2□}$, а так как □ = □□$_0$ то пропускная способность канала □ = □□$^{-2□}$.
Максимальная пропускная способность достигается при G = 0,5 и при $S = \frac{1}{2e}$, что составляет примерно 18% номинальной пропускной способности системы.
Слотированная ALOHA.
Модификация чистой ALOHA, в которой все время работы канала разделяется на слоты. Размер слота при этом должен быть равен максимальному времени кадра. Ясно, что такая организация работы канала требует синхронизации. Кто-то, например одна из станций, испускает сигнал начала очередного слота. Поскольку передачу теперь можно начинать не в любой момент, а только по специальному сигналу, то время на обнаружение коллизии сокращается вдвое. Откуда □ = □□$^{-□}$. Максимум пропускной способности слотированной ALOHA наступает при G = 1, где S = $S = \frac{1}{e}$, т.е. составляет около 37% , что вдвое больше, чем у чистой ALOHA. Рассмотрим как G влияет на пропускную способность системы. Для этого подсчитаем вероятность успешной передачи кадра за k попыток. Так как □$^{-□}$ – это вероятность отсутствия коллизии при передаче, то вероятность того, что кадр будет передан ровно за k попыток, можно записать в виде $P_k = e^{-G}(1 - e^{-G})^{k-1}$
Среднее ожидаемое число повторных передач: $E = \sum_{k=1}^{\infty} kP_k = \sum_{k=1}^{\infty} ke^{-G}(1 - e^{-G})^{k-1}$
Эта экспоненциальная зависимость показывает, что с ростом G резко возрастает число повторных попыток, поэтому незначительное увеличение загрузки канала ведет к резкому падению его пропускной способности.
Протоколы множественного доступа с обнаружением несущей.
Протоколы, реализующие идею начала передачи только после определения, занят канал или нет, называются **протоколами с обнаружением несущей** – CSMA (Carrier Sensitive Multiple Access).
Настойчивые и ненастойчивые CSMA-протоколы.
Если канал занят, то станция ждет, а как только он освободился, пытается сразу начать передачу. Если при этом произошла коллизия, станция ожидает случайный промежуток времени и все начинает сначала. Этот протокол называется **настойчивым CSMA-протоколом первого уровня** или **1-настойчивым CSMA-протоколом**, поскольку станция, следуя этому протоколу, начинает передачу с вероятностью 1, как только обнаруживает, что канал свободен. Здесь важную роль играет задержка распространения сигнала в канале. Всегда существует вероятность того, что, как только одна станция начала передачу, другая станция также стала готова передавать. Если вторая станция проверит состояние канала прежде чем до нее дойдет сигнал от первой станции о том, что она заняла канал, то вторая станция сочтет канал свободным и начнет передачу. В результате

возникает коллизия. Чем больше время задержки сигнала, тем больше вероятность такого случая и тем хуже производительность канала.
Однако даже если время задержки сигнала будет равно нулю коллизии все равно могут возникать. Например, если во время передачи готовыми к передаче оказались две станции. В этом случае они подождут, пока ранее начатая передача будет закончена, а затем будут состязаться между собой. Тем не менее этот протокол более эффективен, чем любая из систем ALOHA, так как станция учитывает состояние канала, прежде чем начать действовать.
Другим вариантом CSMA-протокола является **ненастойчивый CSMA-протокол**. Основное отличие его от предыдущего состоит в том, что готовая к передаче станция опрашивает канал. Если канал свободен, то она начинает передачу. Если же канал занят, то она не будет настойчиво его опрашивать в ожидании, когда он освободится, а будет делать это через случайные отрезки времени. Это несколько увеличивает задержку при передаче сигнала для станции, но общая эффективность протокола возрастает. И, наконец, настойчивый CSMA-протокол уровня p. Который применяется квотированным каналам. Когда станция готова к передаче, она опрашивает канал. Если канал он свободен, то она с вероятностью p передает свой кадр и с вероятностью q = 1 – p ждет следующего слота. Так станция действует, пока не передаст кадр. Если во время передачи происходит коллизия, станция ожидает случайный промежуток времени и опрашивает канал снова. Если при опросе он опять оказывается занятым, станция ждет начала следующего слота, и весь алгоритм повторяется.
CSMA-протокол с обнаружением коллизий.
Протоколы этого класса широко используются в локальных сетях. Модель их работы следующая. В момент времени τ_0 одна станция заканчивает передачу очередного кадра, а все другие станции, у которых имеется кадр для передачи, начинают передачу. Естественно, в этом случае происходят коллизии, которые быстро обнаруживаются посредством сравнения отправленного сигнала с тем сигналом, который есть на линии. Обнаружив коллизию, станция сразу прекращает передачу на случайный промежуток времени, после чего все начинается сначала. Таким образом, в работе протокола CSMA/CD можно выделить три стадии: состязания, передачи и ожидания (когда нет кадров для передачи).
Рассмотрим подробнее алгоритм состязаний. Определим, сколько времени станции, начавшей передачу, требуется, чтобы обнаружить коллизию. Обозначим t время распространения сигнала до самой удаленной станции на линии. Для коаксиального кабеля длиной в 1 км t = 5 мкс. В этом случае минимальное время для определения коллизии будет равно 2t. Следовательно, станция не может быть уверена, что она захватила канал до тех пор, пока не убедится, что в течение 2t секунд не было коллизий. Поэтому мы будем рассматривать период состязаний как слотированную систему ALOHA со слотом 2t секунд на один бит. Захватив канал, станция может далее передавать кадр с любой скоростью.
Обнаружение коллизий – это аналоговый процесс, поэтому, чтобы обнаруживать их, необходимо использовать специальные кодировки на физическом уровне.

18. Коммутация пакетов, устройство пакетов. Как устроен и работает пакетный коммутатор (switch). Виды задержек в компьютерной сети и способы управления ими (приоритеты, веса и гарантированная скорость потока). Управление потоком при пакетной коммутации.

Рассмотрим пакетный коммутатор с буферизацией на выходе, как гарантию того, что пакеты из буфера будут обслужены с определенной скоростью.

Пакетные коммутаторы работают по принципу FIFO, однако поступающие пакеты могут быть разной длины, а значит, при равномерной работе, окажется, что кто-то получит бОльшую долю пропускной способности.

Если пакеты имеют равный размер, то время задержки каждого пакета не более, чем размер буфера делить на пропускную способность канала.

Задание строгих приоритетов нарушает справедливость: на каждом цикле работы сети есть пакеты с высоким приоритетом, то они обслуживаются в первую очередь. При этом пакеты с низким приоритетом оказываются «задавлены», в то время как с высоким даже не знают о них. Это плохо.

Взвешенная справедливая очередь (англ. Weighted fair queuing, WFQ) — механизм планирования пакетных потоков данных с различными приоритетами. Его целью является регулировать использования одного канала передачи данных несколькими конкурирующими потоками. В данном случае под потоком понимается очередь пакетов данных.

Это обобщение алгоритма честных планировщиков (англ. Fair Queuing) (FQ). Оба планировщика имеют отдельные FIFO-очереди для каждого потока данных. Так, если канал со скоростью R используется для N потоков, то скорость обработки каждого из них будет R/N при использовании честного планировщика.

Честный планировщик с приоритетными коэффициентами позволяет регулировать долю каждого потока. Если имеется N активных потоков, с приоритетами □1, ..., □□ , то i-ый поток будет иметь скорость:

$$\frac{W_{w_i}}{\sum_{i=1}^{N} w_i}$$

- FIFO очередь – нет приоритетов, не гарантирована скорость.
- Строгие приоритеты: высокоприоритетный трафик «не видит» низкоприоритетного трафика в сети. Полезно, когда высокоприоритетного трафика ограниченное количество.
- Waited Fair Queuing (WFQ) позволяет каждому потоку обеспечить гарантированный сервис, планируя их в порядке bit-by-bit finishing time.

Коммутация пакетов: гарантирование задержки

Основные допущения:

- Пакеты не теряются.
- FIFO обслуживание.

Поскольку мы не можем управлять процессом поступления, его можно ограничить. Пусть число бит, которые могут поступить за период □ ограничены □ + □□ .Где, например, □ = □, □ = □1 – размер канала, тогда мы не столкнемся с проблемой переполнения буфера **(для конкретного случая).**

Одной из причин перегрузок является неравномерный трафик. Если бы этого не было, перегрузок можно было бы избежать. Поэтому используются механизмы формирования трафика, например, **алгоритм текущего ведра.**

Каждая станция, подключенная к сети имеет в своем интерфейсе буфер, подобный ведру и сбрасывающий пакеты при переполнении. Для регулирования скорости поступления пакетов можно, например, использовать системные часы и установить предел числа пакетов, которые можно направить в сеть в промежуток времени. Если пакеты имеют переменную длину – можно ограничивать число байтов, поступающих в сеть.

Иногда бывает полезно ускорить передачу пакетов в сеть, тогда используют **алгоритм текущего ведра с маркерами**. Вместе с пакетами в ведро поступают маркеры, а пакеты выходят только при наличии определенного числа маркеров. Тогда можно накапливать маркеры и кратковременно ускорять передачу пакетов в сеть. Особенность – при переполнении буфера маршрутизатору временно будет запрещено передавать пакеты в сеть.

□ + □□ = □□, тогда $S = \frac{C}{M-\rho}$, где □ – длительность временного увеличения трафика на входе, □ – скорость поступления маркеров Б/с, – максимальная скорость входного трафика Б/с, – емкость корзины в байтах.

Несмотря на то, что технически это возможно, лишь некоторые сети могут управлять e2e задержкой. Причины:

1) Слишком сложно и хлопотно.
2) В большинстве сетей комбинация прогнозирования и приоритетов дает вполне приемлемые результаты.

$$e2e \text{ задержка}, \tau = \sum_i \left(\frac{p}{r_i} + \frac{l_i}{c} + Q_i(t) \right)$$

- Если мы знаем длину очереди и дисциплину ее обслуживания, то мы можем ограничить величину задержки в ней.
- Выбрав длину очереди, и, используя WFQ, мы можем определить скорость обслуживания.
- Поэтому самое главное не допустить сброса пакетов. Для этого можно использовать текущий буфер.
- Таким образом, мы можем ограничить величину e2e задержки.

Управление потоком при пакетной коммутации

В компьютерных сетях неизбежны потери пакетов данных, в частности, из-за переполнения буферной памяти хотя бы одного из узлов, расположенных на пути от источника к приёмнику, включая последний. Такие потери, связанные с переполнениями, в дальнейшем именуются **перегрузками узлов сети**.

Существует множество способов предотвращения и устранения перегрузок; эти способы, в большинстве своем, основаны на управлении потоками данных. Особое место занимает обслуживание пакетов с учетом их приоритетов.

Способ 1. Управление потоком данных регулировкой длительности пауз между пакетами.

Прототип. В процессе передачи данных приемник замечает устойчивую нехватку прибывающих пакетов (например, отслеживая их порядковые номера) и посылает источнику данных управляющий пакет, содержащий команду XOFF приостановки потока данных. Адрес источника данных известен приемнику, так как поступающие к нему пакеты данных содержат информацию об адресах отправителя и адресата. При этом также посылаются запросы на повторную передачу потерянных пакетов.

Получив команду XOFF, источник данных полностью прекращает передачу пакетов и возобновляет ее либо через некоторое время, оговоренное в протоколе обмена данными, либо после получения от приемника команды XON возобновления передачи.

Недостатки:

1) Поток либо есть, либо его нет. Запаздывание выполнения команд может привести к неоправданному простаиванию передатчика и периодическому возникновению новых перегрузок, при которых некоторая часть пакетов, в том числе, принадлежащих другим потокам, будет утеряна.
2) При длительной перегрузке приемник пересылает передатчику серию одинаковых команд приостановки потока, что засоряет канал связи большим количеством повторяющихся служебных пакетов.
3) Команды приостановки работы передатчика формируются приемником только в том случае, когда число отвергнутых в связи с переполнением буфера пакетов достаточно велико.
4) Приостановки работы передатчика увеличивают среднюю и максимальную задержки передачи пакетов по трассе, что может снизить заданные в контракте между пользователем и провайдером показатели качества обслуживания канала.

Решение. Предлагаемое решение в значительной степени устраняет перечисленные недостатки благодаря плавной и «опережающей события» регулировке скорости передачи данных источником. Скорость регулируется изменением длительности пауз между пакетами: чем больше паузы, тем ниже скорость передачи данных, и наоборот. Отметим, что наличие паузы не означает, что сигнал в линии связи отсутствует – сигнал присутствует постоянно, но в нем нет флаговых кодов, обозначающих начало пакета, либо наоборот – передается непрерывный поток этих кодов.

Способ 2. Управление потоком данных оповещением источника пакетов о причинах перегрузки.

Прототип. В ответ на каждый пакет или на группу пакетов приемник посылает источнику ответные пакеты ACK или NACK. Ответ ACK подтверждает успешный прием, ответ NACK служит запросом повторной передачи одного пакета или группы пакетов. Если источник данных при повышении скорости передачи пакетов или на некоторой фиксированной скорости начинает получать чрезмерное число запросов повторной передачи, то, вероятнее всего, по крайней мере, один из узлов трассы вошел в режим перегрузки. В этом случае источник данных резко снижает скорость передачи пакетов или (и) увеличивает их длину, чтобы уменьшить долю передаваемых в потоке данных служебных битов, образующих заголовки. В дальнейшем источник данных постепенно либо методом случайных проб и ошибок повышает скорость передачи данных, продвигаясь к допустимой верхней границе с учетом некоторого разрешенного запаса повышения скорости. Такой способ называют **«медленным стартом»**.

Рассмотренный способ управления потоком данных не предотвращает предстоящую потерю пакетов, а позволяет реагировать только на свершившийся факт перегрузки промежуточного узла сети или приемника данных. В этом состоит его **основной недостаток**.

Другой прототип. Идея состоит в том, чтобы вовремя предупредить источник данных A об угрозе перегрузки одного или нескольких узлов вдоль трассы AB распространения пакетов данных D. Таким предупреждением служит бит Z, входящий в состав заголовка ответного пакета ACK или NACK.

Решение. Поставленная задача решается расширением одноразрядного признака Z до нескольких битов.

Узел может испытывать перегрузку по крайней мере по одной из следующих причин:

1) Сужение полосы (пропускной способности) канала AB из-за появления «узкого места».
    - Увеличился до значительного уровня ранее малозаметный конкурирующий поток данных по маршруту M4M2M3, который использует тот же канал M2M3, что и маршрут AB. В результате узел M2 перераспределил полосу этого канала в ущерб маршруту AB.
    - Узел M2 изменил тип модуляции сигнала в канале M2M3, снизив скорость передачи в связи с ухудшением соотношения «сигнал/шум» в этом канале.
2) Процессор узла M2 по тем или иным причинам перестал справляться с объемом работы по анализу заголовков пакетов, следующих по маршруту AB.

Первая и вторая причины надвигающейся перегрузки отображаются соответственно кодами □ = 01 и □ = 10, отсутствие опасности перегрузки соответствует коду □ = 00, обе причины одновременно вызывают формирование кода □ = 11.

Способ 3. Управление потоком данных с компенсацией инерционности контура обратной связи. Плавность управления достигается дроблением серий пакетов и более интеллектуальным алгоритмом формирования команд XON и XOFF возобновления и прекращения передачи потока.

19. Алгоритмы маршрутизации в Интернет: основные подходы. Структура сети Интернет, понятие автономной системы, протокол внешней маршрутизации BGP. Явление перегрузки и основные методы борьбы с ней. Перегрузка: AIMD в случае одного потока и в случае нескольких потоков.

Общие сведения.

Основной задачей сетевого уровня является маршрутизация пакетов. Пакеты маршрутизируются всегда, т.е. независимо от того, какую внутреннюю организацию имеет транспортная среда: с виртуальными каналами или дейтаграммную. Разница состоит лишь в том, что в первом случае этот маршрут устанавливается один раз для всех пакетов, а во втором – для каждого пакета отдельно. Первый случай называют иногда маршрутизацией сессии, поскольку маршрут устанавливается на все время передачи данных пользователя, т.е. на время сессии.

Алгоритм маршрутизации реализует программное обеспечение маршрутизатора на сетевом уровне, т.е. он отвечает за определение, по какой из линий, доступных маршрутизатору, отправлять пакет дальше. При этом независимо от выбора маршрута (для сессии или для каждого пакета в отдельности) алгоритм маршрутизации должен обладать следующими свойствами: корректностью, простотой, устойчивостью, стабильностью, справедливостью и оптимальностью.

1) Корректность – свойство алгоритма маршрутизации, определяющее, что при любых обстоятельствах этот алгоритм либо найдет маршрут для доставки пакета адресату, либо выдаст сообщение о невозможности его доставки. Третьего варианта быть не может. При этом крайне желательно, чтобы алгоритм также сообщил о причинах невозможности доставки пакета.
2) Простота – свойство, определяющее вычислительную сложность алгоритма маршрутизации: чем она меньше, тем алгоритм проще, и тем меньше ресурсов маршрутизатора тратится на решение задачи маршрутизации.
3) Устойчивость – свойство алгоритма маршрутизации сохранять работоспособность независимо от каких-либо сбоев, отказов в системе передачи данных или транспортной среде, а также изменений топологии (отключения хостов или машин транспортной среды, разрушения каналов и т.п.). Алгоритм маршрутизации должен адаптироваться ко всем таким изменениям, не требуя при этом перезагрузки транспортной среды или остановки абонентских машин.
4) Стабильность – весьма важное свойство алгоритма маршрутизации. Существуют алгоритмы, которые никогда не приводят к какому-либо определенному маршруту, как бы долго они ни работали. Это означает, что адаптация алгоритма к изменениям в топологии иди конфигурации транспортной среды мажет оказаться весьма продолжительной, и более того, она может оказаться сколь угодно долгой.
5) Справедливость – свойство, означающее, что все пакеты независимо оттого, из какого канала они поступили, будут обслуживаться маршрутизатором равномерно, т.е. никакому направлению не будет отдаваться предпочтение, для всех абонентов будет всегда выбираться оптимальный маршрут.

Следует отметить, что справедливость и оптимальность часто могут вступать в противоречие друг с другом при неудачном выборе критерия оптимизации.

Прежде чем искать компромисс между оптимальностью и справедливостью, необходимо решить, что является критерием оптимизации маршрута. Один из возможных критериев – средняя задержка пакета (обратите внимание, что именно средняя задержка).

Другой критерий – пропускная способность транспортной среды. Однако эти критерии конфликтуют. Согласно теории массового обслуживания, если система с очередями функционирует близко к своему насыщению, то задержка в очереди увеличивается. Как компромисс во многих сетях минимизируется число переходов между маршрутизаторами. Один такой переход называется скачком, или переходом (hop). Уменьшение числа скачков сокращает маршрут, а следовательно, сокращает задержку и минимизирует необходимую пропускную способность СПД для передачи пакета.

Алгоритмы маршрутизации можно разбить на два больших класса: адаптивные и неадаптивные. Неадаптивные алгоритмы не принимают в расчет текущую загрузку сети и ее текущую топологию. Все возможные маршруты вычисляются заранее и загружаются в маршрутизаторы при загрузке сети. Такая маршрутизация называется статической.

Адаптивные алгоритмы, наоборот, определяют маршрут исходя из текущей загрузки и топологии транспортной среды. Адаптивные алгоритмы различаются способом получения информации (локально от соседних маршрутизаторов или глобально от всех маршрутизаторов), временем изменения маршрута (через каждые T секунд либо только когда изменяется нагрузка, либо когда изменяется топология) и метрикой, используемой при оптимизации (расстояние, число скачков, ожидаемое время передачи и т.н.).

Свойство оптимального пути.

Обоснуем одно важное предположение о свойстве оптимального маршрута, которое будет использоваться в дальнейших рассуждениях. Это свойство состоит в том, что если маршрутизатор J

находится на оптимальном пути между маршрутизаторами I и K, то оптимальный маршрут между J и K принадлежит этому оптимальному пути. Это так, поскольку существование между J и K оптимального маршрута, отличного от части маршрута между I и K, противоречило бы утверждению об оптимальности маршрута между I и K.
Поскольку дерево захода – это дерево, то там нет циклов, и каждый пакет будет доставлен за конечное число шагов. На практике же все может оказаться сложнее: маршрутизаторы могут выходить из строя, могут появляться новые маршрутизаторы, каналы могут выходить из строя, разные маршрутизаторы могут узнавать об этих изменениях в разное время и т.д.
Фактически, Интернет состоит из множества локальных и глобальных сетей, принадлежащих различным компаниям и предприятиям, работающих по самым разнообразным протоколам, связанных между собой различными линиями связи, физически передающих данные по телефонным проводам, оптоволокну, через спутники и радиомодемы. Структура Интернет напоминает паутину, в узлах которой находятся компьютеры, связанные между собой линиями связи. Узлы Интернет, связанные высокоскоростными линиями связи, составляют базис Интернет. Оцифрованные данные пересылаются через маршрутизаторы, которые соединяют сети с помощью сложных алгоритмов, выбирая маршруты для информационных потоков.
Каждый компьютер в Интернет имеет свой уникальный адрес – IP-адрес. Этот номер может быть постоянно закреплен за компьютером или же присваиваться динамически – в тот момент, когда пользователь соединился с провайдером, но в любой момент времени в Интернет не существует двух компьютеров с одинаковыми IP-адресами.
Доменное имя – это уникальное имя, которое данный поставщик услуг избрал себе для идентификации. Для преобразования имени в адрес компьютер посылает запрос серверу DNS, начиная с правой части доменного имени и двигаясь влево.
Данные в Интернет пересылаются не целыми файлами, а небольшими блоками, которые называются пакетами. Каждый пакет содержит в себе адреса компьютеров отправителя и получателя, передаваемые данные и порядковый номер пакета в общем потоке данных. Благодаря тому, что каждый пакет содержит все необходимые данные, он может доставляться независимо от других, и довольно часто случается так, что пакеты добираются до места назначения разными путями. А компьютер-получатель затем выбирает из пакетов данные и собирает из них тот файл, который был заказан.
Порт – это число, которое добавляется к адресу компьютера, которое указывает на программу, для которой данные предназначены.
В Интернет используются не просто доменные имена, а универсальные указатели ресурсов URL (Universal Resource Locator).
URL включает в себя:

- метод доступа к ресурсу, т.е. протокол доступа (http, gopher, WAIS, ftp, file, telnet и др.);
- сетевой адрес ресурса (имя хост-машины и домена);
- полный путь к файлу на сервере.

Сервер в сети Интернет – это компьютер, обеспечивающий обслуживание пользователей сети: разделяемый доступ к дискам, файлам, принтеру, системе электронной почты. Обычно сервер – это совокупность аппаратного и программного обеспечения.
Сайт – обобщенное название совокупности документов в Интернет, связанных между собой ссылками.
Шлюз (gateway) – это компьютер или система компьютеров со специальным программным обеспечением, позволяющая связываться двум сетям с разными протоколами.
Домашняя страница – это персональная Web-страница конкретного пользователя или организации.
Автономная система (AS) в интернете – это система IP-сетей и маршрутизаторов, управляемых одним или несколькими операторами, имеющими единую политику маршрутизации с Интернетом.
Протокол BGP используется для передачи информации о внутренних маршрутах между автономными системами. Протокол BGP может быть использован для определения различных типов маршрутов:

- Inter-autonomous system routing – маршруты, которые соединяют данную автономную систему с одной или несколькими другими автономными системами.
- Intra-autonomous system routing – протокол может быть использован для определения маршрута внутри автономной системы, в том случае, когда несколько маршрутизаторов участвуют в процессе определения маршрута BGP.
- Pass-through autonomous system – протокол может быть использован для определения маршрутов, которые проходят через автономную систему, которая не участвует в процессе BGP.

Для обеспечения информационного обмена маршрутизаторы BGP используют сообщения стандартной формы. Для передачи этих сообщений в протоколе BGP предусматривается использование транспортного протокола TCP. Сообщения BGP передаются в следующих случаях:

- Начало сеанса (Open).
- Для периодической проверки состояния соседа (Keep Alive).
- При изменении содержания таблицы маршрутов автономной системы (Update).

- При возникновении аварийной ситуации (Notification).

Формат сообщения BGP. Каждое сообщение BGP состоит из заголовка и последующих специфических полей:

MARKER	
MARKER	
MARKER	
MARKER	
LENGTH	TYPE

В поле LENGTH помещается размер сообщения (вместе с заголовком), выраженный в байтах. В поле TYPE помещается код сообщения в соответствии со следующей таблицей:

TYPE	Сообщение
1	OPEN
2	UPDATE
3	NOTIFICATION
4	KEEPALIVE

В поле маркера может быть помещена информация, которая необходима для выполнения операции аутентификации абонента. Если установление подлинности абонента не требуется, маркер формируется значениями – все «1».

OPEN – первое сообщение, которое должно быть передано маршрутизатором BGP после установления соединения TCP.

UPDATE используется для представления маршрута соседнему маршрутизатору BGP. Это сообщение одновременно может быть использовано для уничтожения маршрутов, которые перестали существовать.

Когда в транспортной среде находится в одно и то же время слишком много пакетов, ее производительность начинает падать. Перегрузка может возникнуть в силу нескольких причин. Например, если сразу несколько потоков, поступающих по нескольким входным линиям, устремятся на одну и ту же выходную линию. Если буфер маршрутизатора переполнится, то пакеты начнут теряться. Перегрузки могут случаться и из-за недостаточной скорости процессора. Если процессор будет не в состоянии справиться своевременно с рутинными задачами (размещения пакета в буфере, корректировка таблиц и т.п.), то даже при наличии линий с достаточной пропускной способностью очередь будет расти. Аналогичная картина может случиться при быстром процессоре, но медленном канале и наоборот. Таким образом, источник проблемы – несбалансированность производительности компонентов системы. Перегрузка – это глобальная проблема в сети.

Управление перегрузками – это такая организация потоков в транспортной среде, при которой потоки соответствуют пропускной способности подсети и не превышают ее.

Основные принципы управления перегрузками.

В терминологии теории управления все методы управления перегрузками в сетях можно разбить на две большие группы: с открытым контуром управления и закрытым контуром управления. Методы с открытым контуром предполагают, что все продумано и предусмотрено заранее в конструкции системы, и если нагрузка находится в заданных пределах, то перегрузки не происходит. Если же нагрузка начинает превышать определенные пределы, то заранее известно, когда и где начнется сброс пакетов, в каких точках сети начнется перепланировка ресурсов, и т.п. Главное, что все эти меры будут приниматься вне зависимости от текущего состояния сети.

Решения, основанные на закрытом контуре, используют обратную связь. Эти решения включают три этапа:

- Наблюдение за системой для определения, где и когда началась перегрузка.
- Передача данных туда, где будут предприняты надлежащие меры.
- Перестройка функционирования системы для устранения проблемы.

При наблюдении за системой используются разные метрики для определения перегрузки. Основными среди них являются:

- Процент пакетов, сброшенных из-за нехватки памяти в буферах.
- Средняя длина очередей в системе.
- Число пакетов, для которых наступил time_out и для которых были сделаны повторные передачи.
- Средняя задержка пакета при доставке и среднее отклонение задержки при доставке пакета.

Следующий шаг при использовании обратной связи – передать информацию о перегрузке туда, где что-то может быть сделано, чтобы исправить положение. Например, маршрутизатор, обнаруживший перегрузку, может направить сообщение о перегрузке всем источникам сообщений. Ясно, что это увеличит нагрузку в сети, причем именно в тот момент, когда это менее всего желательно. Однако есть и другие возможности. Например, в каждом пакете зарезервировать специальный бит перегрузки, и если какой-то маршрутизатор обнаружил перегрузку, то он устанавливает этот бит, тем самым сообщая другим о ней (вспомним структуру кадра во Frame

Relay). Другое решение напоминает прием, используемый некоторыми радиостанциями: направлять несколько автомашин по дорогам, чтобы обнаруживать пробки, а затем сообщать о них по радиоканалам, предупреждая другие машины, призывая их пользоваться объездными путями. По аналогии с этим решением в сети рассылаются специальные пробные пакеты, которые проверяют нагрузку, и если где-то обнаружена перегрузка, то о ней сообщается всем и происходит перенаправление пакетов так, чтобы обогнуть перегруженные участки.

Методы, предотвращающие перегрузки.

Рассмотрение методов, предотвращающих перегрузки, начнем с методов для систем с открытым контуром. Эти методы ориентированы на минимизацию перегрузок при первых признаках их проявлений, а не на борьбу с перегрузками, когда они уже случились. Основные факторы, влияющие на перегрузки на канальном, сетевом и транспортном уровнях, перечислены в таблице:

Уровень	Факторы
Транспортный	Повторная передача Порядок передачи бит Уведомления Управление потоком Значение timeout
Сетевой	Виртуальные каналы vs. дейтаграммы внутри подсети Очередность пакетов и сервисы Сброс пакета Алгоритм маршрутизации Управление временем жизни пакетов
Канальный	Повторная передача Порядок передачи бит Уведомления Управление потоком

Методы:

1) Схема управления потоком (небольшое окно) сдерживает нарастание трафика и предотвращает появление перегрузок.
2) Методы управления очередями, организация очередей: одна общая на входе или одна общая на выходе; по одной на каждую входную линию или на каждую выходную; по одной очереди на каждую входную и выходную – все это влияет на появление перегрузок.
3) Выбор метода сброса пакетов также влияет на перегрузки. Правильная маршрутизация, равномерно использующая каналы в транспортной среде, позволяет избежать перегрузки.
4) Методы, регулирующие время жизни пакета в сети, также влияют на образование перегрузок.

Additive-increase/multiplicative-decrease (AIMD) алгоритм является алгоритмом управления перегрузками с обратной связью. AIMD сочетает линейный рост окна с экспоненциальным сокращением, когда происходит сброс пакета.

Пусть W – размер окна, тогда:

- Если пакет получен, то: $W = W + \frac{1}{W}$
- Если пакет сброшен, то: $W = \frac{W}{2}$

AIMD в случае одного потока:

1) Окно увеличивают, сокращают в соответствии с AIMD, можно определить как много байт канал еще может вместить.
2) Пилообразное поведение графика размера окна от времени – это нормальная форма динамики.
3) Скорость отправки постоянная: $R = \frac{W}{RTT}$. (RTT – Round-trip time), если есть достаточно буферного пространства (R * RTT > W).

AIMD в случае нескольких потоков:

1) Окно увеличивают, сокращают в соответствии с AIMD.
2) В «узком месте» будут скапливаться пакеты разных потоков.
3) Скорость отправки меняется в зависимости от размера окна.
   Доля теряемых пакетов:

$$p = \frac{1}{A}, \text{где } A = \frac{3}{8}W_{max}^2, R = \frac{A}{\frac{W_{max}}{2}RTT}.$$

$$R = \sqrt{\frac{3}{2}}\frac{1}{RTT\sqrt{p}}$$

4) AIMD очень чувствителен к частоте потери пакетов.
5) AIMD ущемляет потоки с большим RTT.

20. Качество программного обеспечения и методы его контроля. Тестирование и другие методы верификации.

Качество программного обеспечения определяется в стандарте ISO 9126 как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

Различаются понятия ***внутреннего качества***, связанного с характеристиками ПО самого по себе, без учета его поведения; ***внешнего качества***, характеризующего ПО с точки зрения его поведения; и ***качества ПО при использовании*** в различных контекстах — того качества, которое ощущается пользователями при конкретных сценариях работы ПО. Для всех этих аспектов качества введены метрики, позволяющие оценить их. Кроме того, для создания добротного ПО существенно *качество технологических процессов* его разработки.

Верификация обозначает проверку того, что ПО разработано в соответствии со всеми требованиями к нему, или что результаты очередного этапа разработки соответствуют ограничениям, сформулированным на предшествующих этапах.

Валидация — это проверка того, что сам продукт правилен, т.е. подтверждение того, что он действительно удовлетворяет потребностям и ожиданиям пользователей, заказчиков и других заинтересованных сторон.

Эффективность верификации и валидации, как и эффективность разработки ПО в целом, зависит от полноты и корректности формулировки требований к программному продукту.

Основой любой системы обеспечения качества являются методы его обеспечения и контроля.

Методы обеспечения качества представляют собой техники, гарантирующие достижение определенных показателей качества при их применении. Мы будем рассматривать подобные методы на протяжении всего курса.

Методы контроля качества позволяют убедиться, что определенные характеристики качества ПО достигнуты. Сами по себе они не могут помочь их достижению, они лишь помогают определить, удалось ли получить в результате то, что хотелось, или нет, а также найти ошибки, дефекты и отклонения от требований. Методы контроля качества ПО можно классифицировать следующим образом.

1) Методы и техники, связанные с выяснением свойств ПО во время его работы. Это, прежде всего, все виды *тестирования*, а также *профилирование* и измерение количественных показателей качества, которые можно определить по результатам работы ПО — эффективности по времени и другим ресурсам, надежности, доступности и пр.
2) Методы и техники определения показателей качества на основе симуляции работы ПО с помощью моделей разного рода. К этому виду относятся *проверка на моделях (model checking)*, а также *прототипирование (макетирование)*, используемое для оценки качества принимаемых решений.
3) Методы и техники, нацеленные на выявление нарушений формализованных правил построения исходного кода ПО, проектных моделей и документации. К методам такого рода относится *инспектирование кода*, заключающееся в целенаправленном поиске определенных дефектов и нарушений требований в коде на основе набора шаблонов, автоматизированные методы поиска ошибок в коде, не основанные на его выполнении, методы проверки документации на согласованность и соответствие стандартам.
4) Методы и техники обычного или формализованного анализа проектной документации и исходного кода для выявления их свойств. К этой группе относятся многочисленные методы *анализа архитектуры ПО*, о которых пойдет речь в следующей лекции, методы формального доказательства свойств ПО и формального анализа эффективности применяемых алгоритмов.

Тестирование — это проверка соответствия ПО требованиям, осуществляемая с помощью наблюдения за его работой в специальных, искусственно построенных ситуациях. Такого рода ситуации называют тестовыми или просто ***тестами***.

Тестирование — конечная процедура. Набор ситуаций, в которых будет проверяться тестируемое ПО, всегда конечен. Более того, он должен быть настолько мал, чтобы тестирование можно было провести в рамках проекта разработки ПО, не слишком увеличивая его бюджет и сроки. Это означает, что при тестировании всегда проверяется очень небольшая доля всех возможных ситуаций. По этому поводу Дейкстра заметил, что тестирование позволяет точно определить, что в программе есть ошибка, но не позволяет утверждать, что там нет ошибок.

Тем не менее, тестирование может использоваться для достаточно уверенного вынесения оценок о качестве ПО. Для этого необходимо иметь ***критерии полноты тестирования***, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними. Этот критерий может утверждать, что все равно в какой из ситуаций, A или B, проверять правильность работы ПО, или, если программа правильно работает в ситуации A, то, скорее всего, в ситуации B все тоже будет правильно. Часто критерий полноты тестирования задается при помощи определения эквивалентности ситуаций, дающей конечный набор классов ситуаций. В этом случае считают, что все равно, какую из ситуаций одного класса использовать в качестве теста. Такой критерий называют ***критерием тестового покрытия***, а процент классов эквивалентности ситуаций, случившихся во время тестирования, — достигнутым ***тестовым покрытием***.

На основе исходных данных, используемых для построения тестов, тестирование делят на следующие виды.

Тестирование черного ящика, нацеленное на проверку требований. Тесты для него и критерий полноты тестирования строятся на основе требований и ограничений, четко зафиксированных в спецификациях, стандартах, внутренних нормативных документах. Часто такое тестирование называется ***тестированием на соответствие (conformance testing).*** Частным случаем его является ***функциональное тестирование*** — тесты для него, а также используемые критерии полноты проведенного тестирования определяют на основе требований к функциональности. Еще одним примером тестирования на соответствие является ***аттестационное*** или ***квалификационное тестирование***, по результатам которого программная система получает (или не получает) официальный документ, подтверждающий ее соответствие определенным требованиям и стандартам.

Тестирование белого ящика, оно же ***структурное тестирование*** — тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Критерии полноты основаны на проценте элементов кода, которые отработали в ходе выполнения тестов. Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о связи требований с определенными ограничениями на значения внутренних данных системы (например, на значения параметров вызовов, результатов и локальных переменных).

Тестирование, нацеленное на определенные ошибки. Тесты для такого тестирования строятся так, чтобы гарантированно выявлять определенные виды ошибок. Полнота тестирования определяется на основе количества проверенных ситуаций по отношению к общему числу ситуаций, которые мы пытались проверить. К этому виду относится, например, ***тестирование на отказ (smoke testing)***, в ходе которого просто пытаются вывести систему из строя, давая ей на вход как обычные данные, так и некорректные, с нарочно внесенными ошибками. Другим примером служит метод оценки полноты тестирования при помощи набора ***мутантов*** — программ, совпадающих с тестируемой всюду, кроме нескольких мест, где специально внесены некоторые ошибки. Чем больше мутантов не проходит успешно через данный набор тестов, тем полнее и качественнее проводимое с его помощью тестирование. Еще одна классификация видов тестирования основана на том уровне, на который оно нацелено. Эти же разновидности тестирования можно связать с фазой жизненного цикла, на которой они выполняются.

1) ***Модульное тестирование (unit testing)*** предназначено для проверки правильности отдельных модулей, вне зависимости от их окружения. При этом проверяется, что если модуль получает на вход данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны. Для описания критериев корректности входных и выходных данных часто используют ***программные контракты — предусловия***, описывающие для каждой операции, на каких входных данных она предназначена работать, ***постусловия***, описывающие для каждой операции, как должны соотноситься входные данные с возвращаемыми ею результатами, и ***инварианты***, определяющие критерии целостности внутренних данных модуля. Модульное тестирование является важной составной частью ***отладочного тестирования***, выполняемого разработчиками для отладки написанного ими кода.
2) ***Интеграционное тестирование (integration testing)*** предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая взаимных ограничений на такое взаимодействие, например, предусловий вызываемых операций. Интеграционное тестирование также используется при отладке, но на более позднем этапе разработки.
3) ***Системное тестирование (system testing)*** предназначено для проверки правильности работы системы в целом, ее способности правильно решать поставленные пользователями задачи в различных ситуациях. Системное тестирование выполняется через внешние интерфейсы ПО и тесно связано с ***тестированием пользовательского интерфейса*** (или через пользовательский интерфейс), проводимым при помощи имитации действий пользователей над элементами этого интерфейса. Частными случаями этого вида тестирования являются ***тестирование графического пользовательского интерфейса*** (Graphical User Interface, GUI) и ***пользовательского интерфейса Web-приложений*** (WebUI). Если интеграционное и модульное тестирование чаще всего проводят, воздействуя на компоненты системы при помощи операций предоставляемого ими программного интерфейса (Application Programming Interface, API), то на системном уровне без использования пользовательского интерфейса не обойтись, хотя тестирование через API в этом случае также вполне возможно.

Основной недостаток тестирования состоит в том, что проводить его можно, только когда проверяемый элемент программы уже разработан. Снизить влияние этого ограничения можно, подготавливая тесты (а это — наиболее трудоемкая часть тестирования) на основе требований заранее, когда исходного кода еще нет. Подход опережающей разработки тестов с успехом используется, например, в рамках XP.

Проверка на моделях ***Проверка свойств на моделях (model checking)*** — проверка соответствия ПО требованиям при помощи формализации проверяемых свойств, построения формальных моделей проверяемого ПО (чаще всего в виде автоматов различных видов) и автоматической проверки выполнения этих свойств на построенных моделях. Проверка свойств на моделях позволяет проверять достаточно сложные свойства автоматически, при минимальном участии человека. Однако она оставляет открытым вопрос о том, насколько выявленные свойства модели можно переносить на само ПО. Обычно при помощи проверки свойств на моделях анализируют два вида свойств алгоритмов, использованных при построении ПО. ***Свойства безопасности (safety properties)*** утверждают, что нечто нежелательное никогда не случится в ходе работы ПО. ***Свойства живучести (liveness properties)*** утверждают, наоборот, что нечто желательное при любом развитии событий произойдет в ходе его работы. Примером свойства первого типа служит отсутствие *взаимных блокировок (deadlocks)*. Взаимная блокировка возникает, если каждый из группы параллельно работающих в проверяемом ПО процессов или потоков ожидает прибытия данных или снятия блокировки ресурса от одного
из других, а тот не может продолжить выполнение, ожидая того же от первого или от третьего процесса, и т.д.
Примером свойства живучести служит гарантированная доставка сообщения, обеспечиваемая некоторыми протоколами — как бы ни развивались события, если сетевое соединение между машинами будет работать, посланное с одной стороны (процессом на первой машине) сообщение будет доставлено другой стороне (процессу на второй машине).
В классическом подходе к проверке на моделях проверяемые свойства формализуются в виде формул так называемых *временны**х** логик*. Их общей чертой является наличие операторов «всегда в будущем» и «когда-то в будущем». Заметим, что второй оператор может быть выражен с помощью первого и отрицания — то, что некоторое свойство когда-то будет выполнено, эквивалентно тому, что отрицание этого свойства не будет выполнено всегда. Свойства безопасности легко записываются в виде «всегда будет выполнено отрицание нежелательного свойства», а свойства живости — в виде «когда-то обязательно будет выполнено желаемое».
Проверяемая программа в классическом подходе моделируется при помощи конечного автомата. Проверка, выполняемая автоматически, состоит в том, что для всех достижимых при работе системы состояний этого автомата проверяется нужное свойство. Если оно оказывается выполненным, выдается сообщение об успешности проверки, если нет — выдается трасса, последовательность выполнения отдельных шагов программы, моделируемых переходами автомата, приводящая из начального состояния в такое, в котором нужное свойство нарушается. Эта трасса используется для анализа происходящего и исправления либо программы, либо модели, если ошибка находится в ней.
Основная проблема этого подхода — огромное, а часто и бесконечное, количество состояний в моделях, достаточно хорошо отражающих поведение реальных программ. Для борьбы с комбинаторным взрывом состояний применяются различные методы оптимизации представления автомата, выделения и поиска состояний, существенных для выполнения проверяемого свойства.

21. Виды параллельной обработки данных, их особенности. Компьютеры с общей и распределенной памятью. Вычислительные кластеры: узлы, коммуникационная сеть, способы построения. Производительность вычислительных систем, методы оценки и измерения.

Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и собственно параллельность. **Параллельная обработка**. Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что есть пять таких же независимых устройств, способных работать одновременно, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично система из N устройств ту же работу выполнит за 1000/N единиц времени. УВЕЛИЧЕНИЕ КОЛИЧЕСТВА НЕЗАВИСИМО РАБОТАЮЩИХ УСТРОЙСТВ.

Конвейерная обработка. Что необходимо для сложения двух вещественных чисел, представленных в форме с плавающей запятой? Целое множество мелких операций таких, как сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т.п. Процессоры первых компьютеров выполняли все эти "микрооперации" для каждой пары аргументов последовательно одна за одной до тех пор, пока не доходили до окончательного результата, и лишь после этого переходили к обработке следующей пары слагаемых. УСЛОЖНИТЬ САМО УСТРОЙСТВО, ЧТОБЫ НА РАЗНЫХ ЭТАПАХ МОГЛИ НАХОДИТЬСЯ РАЗНЫЕ ДАННЫЕ. Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передавал бы результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется за одну единицу времени. Если есть одно неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если каждую микрооперацию выделить в отдельный этап (или иначе говорят – ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находится первые пять пар аргументов, а весь набор из ста пар будет обработан за 5+99=104 единицы времени – ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера). ТЕ СУЩЕСТВУЕТ НЕКОТОРАЯ ЗАДЕРЖКА ДЛЯ ТОГО ЧТОБЫ ЗАПОЛНИТЬ ВСЕ ЭТАПЫ КОНВЕЕРА, НО КОГДА ОНА ЗАПОЛНЕНА ДАЛЬШЕ ПРОИСХОДИТ УСКОРЕНИЕ ОБРАБОТКИ.

1. Векторно-конвейерные компьютеры. Конвейерные функциональные устройства и набор векторных команд – это две особенности таких машин. В отличие от традиционного подхода, векторные команды оперируют целыми массивами независимых данных, что позволяет эффективно загружать доступные конвейеры, т.е. команда вида A=B+C может означать сложение двух массивов, а не двух чисел. Характерным представителем данного направления является семейство векторно-конвейерных компьютеров CRAY куда входят, например, CRAY EL, CRAY J90, CRAY T90 (в марте 2000 года американская компания TERA перекупила подразделение CRAY у компании Silicon Graphics, Inc.). 2. Массивно-параллельные компьютеры с распределенной памятью. Идея построения компьютеров этого класса тривиальна: возьмем серийные микропроцессоры, снабдим каждый своей локальной памятью, соединим посредством некоторой коммуникационной среды – вот и все. Достоинств у такой архитектуры масса: если нужна высокая производительность, то можно добавить еще процессоров, если ограничены финансы или заранее известна требуемая вычислительная мощность, то легко подобрать оптимальную конфигурацию и т.п. Однако есть и решающий "минус", сводящий многие "плюсы" на нет. Дело в том, что межпроцессорное взаимодействие в компьютерах этого класса идет намного медленнее, чем происходит локальная обработка данных самими процессорами. Именно поэтому написать эффективную программу для таких компьютеров очень сложно, а для некоторых алгоритмов иногда просто невозможно. К данному классу можно отнести компьютеры Intel Paragon, IBM SP1, Parsytec, в какой-то степени IBM SP2 и CRAY T3D/T3E, хотя в этих компьютерах влияние указанного минуса значительно ослаблено. К этому же классу можно отнести и сети компьютеров, которые все чаще рассматривают как дешевую альтернативу крайне дорогим суперкомпьютерам. 3. Параллельные компьютеры с общей памятью. Вся оперативная память таких компьютеров разделяется несколькими одинаковыми процессорами. Это снимает проблемы предыдущего класса, но добавляет новые – число процессоров, имеющих доступ к общей памяти, по чисто техническим причинам нельзя сделать большим. В данное направление входят многие современные многопроцессорные SMP-компьютеры или, например, отдельные узлы компьютеров HP Exemplar и Sun StarFire. 4. Последнее направление, строго говоря, не является самостоятельным, а скорее представляет собой комбинации предыдущих трех. Из нескольких процессоров (традиционных или векторно-конвейерных) и общей для них памяти сформируем вычислительный узел. Если полученной вычислительной мощности не достаточно, то объединим несколько узлов высокоскоростными каналами. Подобную архитектуру называют кластерной, и по такому принципу построены CRAY SV1, HP Exemplar, Sun StarFire, NEC SX-5, последние модели IBM SP2 и другие. Именно это

направление является в настоящее время наиболее перспективным для конструирования компьютеров с рекордными показателями производительности.
Различных вариантов построения кластеров очень много. Одно из существенных различий лежит в используемой сетевой технологии, выбор которой определяется, прежде всего, классом решаемых задач.
Какими же числовыми характеристиками выражается производительность коммуникационных сетей в кластерных системах? Необходимых пользователю характеристик две: латентность и пропускная способность сети. *Латентность* — это время начальной задержки при посылке сообщений. *Пропускная способность сети* определяется скоростью передачи информации по каналам связи. Если в программе много маленьких сообщений, то сильно скажется латентность. Если сообщения передаются большими порциями, то важна высокая пропускная способность каналов связи. Из-за латентности максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной.
На практике пользователям не столько важны заявляемые производителем пиковые характеристики, сколько реальные показатели, достигаемые на уровне приложений. После вызова пользователем функции посылки сообщения Send() сообщение последовательно проходит через целый набор слоев, определяемых особенностями организации программного обеспечения и аппаратуры. Этим, в частности, определяются и множество вариаций на тему латентности реальных систем. Установили MPI на компьютере плохо, латентность будет большая, купили дешевую сетевую карту от неизвестного производителя, ждите дальнейших сюрпризов.
В заключение параграфа давайте попробуем и для данного класса компьютеров выделить факторы, снижающие производительность вычислительных систем с распределенной памятью на реальных программах. Начнем с уже упоминавшегося ранее *закона Амдала.* Для компьютеров данного класса он играет очень большую роль. В самом деле, если предположить, что в программе есть лишь 2% последовательных операций, то рассчитывать на более чем 50-кратное ускорение работы программы не приходится. Теперь попробуйте критически взглянуть на свою программу. Скорее всего, в ней есть инициализация, операции ввода/вывода, какие-то сугубо последовательные участки. Оцените их долю на фоне всей программы и на мгновенье предположите, что вы получили доступ к вычислительной системе из 1000 процессоров. После вычисления верхней границы для ускорения программы на такой системе, думаем, станет ясно, что недооценивать влияние закона Амдала никак нельзя. Поскольку компьютеры данного класса имеют распределенную память, то взаимодействие процессоров между собой осуществляется с помощью передачи сообщений. Отсюда два других замедляющих фактора — *латентность* и *скорость передачи данных* по каналам коммуникационной среды. В зависимости от коммуникационной структуры программы степень влияния этих факторов может сильно меняться.
Если аппаратура или программное обеспечение не поддерживают возможности *асинхронной посылки сообщений* на фоне вычислений, то возникнут неизбежные накладные расходы, связанные с ожиданием полного завершения взаимодействия параллельных процессов.
Для достижения эффективной параллельной обработки необходимо добиться *максимально равномерной загрузки всех процессоров.* Если равномерности нет, то часть процессоров неизбежно будет простаивать, ожидая остальных, хотя в это время они вполне могли бы выполнять полезную работу. Данная проблема решается проще, если вычислительная система однородна. Очень большие трудности возникают при переходе на неоднородные системы, в которых есть значительное различие либо между вычислительными узлами, либо между каналами связи. Существенный фактор — это *реальная производительность одного процессора* вычислительной системы. Разные модели микропроцессоров могут поддерживать несколько уровней кэш-памяти, иметь специализированные функциональные устройства и т. п. Возьмем хотя бы иерархию памяти компьютера Cray T3E: регистры процессора, кэш- память 1-го уровня, кэш-память 2-го *уровня,* локальная память процессора, *удаленная память другого* процессора. Эффективное использование такой структуры требует особого внимания *при выборе подхода к решению задачи.* Дополнительно каждый микропроцессор может иметь элементы векторно-конвейерной архитектуры. В этом случае ему будут присущи многие факторы, которые мы обсуждали в конце *§3.2.* К сожалению, как и прежде, на работе каждой конкретной программы в той или иной мере сказываются все эти факторы. Однако в отличие от компьютеров других классов, суммарное воздействие изложенных здесь факторов может снизить реальную производительность не в десятки, а в сотни и даже тысячи раз по сравнению с пиковой. Потенциал компьютеров этого класса огромен, добиться на них можно очень многого. Крайняя точка — Интернет. Его тоже можно рассматривать как компьютер с распределенной памятью. Причем, как самый мощный в мире компьютер.

22. Закон Амдала, его следствия. Граф алгоритма. Критический путь графа алгоритм, ярусно-параллельная форма графа алгоритма. Этапы решения задач на параллельных вычислительных системах.

Предположим, что в вашей программе доля операций, которые нужно выполнять последовательно, равна f, где 0<=f<=1 (при этом доля понимается не по статическому числу строк кода, а по числу операций в процессе выполнения). Крайние случаи в значениях f соответствуют полностью параллельным (f=0) и полностью последовательным (f=1) программам. Так вот, для того, чтобы оценить, какое ускорение S может быть получено на компьютере из 'p' процессоров при данном значении f, можно воспользоваться законом Амдала:

S<=1/(f+(1-f)/p)

Если 9/10 программы исполняется параллельно, а 1/10 по-прежнему последовательно, то ускорения более, чем в 10 раз получить в принципе невозможно вне зависимости от качества реализации параллельной части кода и числа используемых процессоров (ясно, что 10 получается только в том случае, когда время исполнения параллельной части равно 0).

Посмотрим на проблему с другой стороны: а какую же часть кода надо ускорить (а значит и предварительно исследовать), чтобы получить заданное ускорение? Ответ можно найти в **следствии из закона Амдала**: для того чтобы ускорить выполнение программы в *q* раз необходимо ускорить не менее, чем в *q* раз не менее, чем (1-1/*q*)-ю часть программы. Следовательно, если есть желание ускорить программу в 100 раз по сравнению с ее последовательным вариантом, то необходимо получить не меньшее ускорение не менее, чем на 99.99% кода, что почти всегда составляет значительную часть программы! Отсюда первый вывод – прежде, чем переделывать код для перехода на параллельный надо подумать. Если оценив заложенный в программе алгоритм вы поняли, что доля последовательных операций велика, то на значительное ускорение рассчитывать не приходится и нужно думать о замене отдельных компонент алгоритма. В ряде случаев последовательный характер алгоритма изменить не так сложно. Допустим, что в программе есть следующий фрагмент для вычисления суммы n чисел:

```
s=0
    Do i = 1, n
        s = s + a(i)
    EndDo
```

(можно то же самое на любом другом языке) По своей природе он строго последователен, так как на i-й итерации цикла требуется результат с (i-1)-й и все итерации выполняются одна за другой. Имеем 100% последовательных операций, а значит и никакого эффекта от использования параллельных компьютеров. Вместе с тем, выход очевиден. Поскольку в большинстве реальных программ (вопрос: а почему в большинстве, а не во всех?) нет существенной разницы, в каком порядке складывать числа, выберем иную схему сложения. Сначала найдем сумму пар соседних элементов: a(1)+a(2), a(3)+a(4), a(5)+a(6) и т.д. Заметим, что при такой схеме все пары можно складывать одновременно! На следующих шагах будем действовать абсолютно аналогично, получив вариант параллельного алгоритма. Казалось бы в данном случае все проблемы удалось разрешить. Но представьте, что доступные вам процессоры разнородны по своей производительности. Значит будет такой момент, когда кто-то из них еще трудится, а кто-то уже все сделал и бесполезно простаивает в ожидании. Если разброс в производительности компьютеров большой, то и эффективность всей системы при равномерной загрузке процессоров будет крайне низкой. Но пойдем дальше и предположим, что все процессоры одинаковы. Проблемы кончились? Опять нет! Процессоры выполнили свою работу, но результат-то надо передать другому для продолжения процесса суммирования... а на передачу уходит время... и в это время процессоры опять простаивают...

Словом, заставить параллельную вычислительную систему или супер-ЭВМ работать с максимальной эффективность на конкретной программе это, прямо скажем, задача не из простых, поскольку **необходимо тщательное согласование структуры программ и алгоритмов с особенностями архитектуры параллельных вычислительных систем.**

Суперлинейное ускорение. При увеличении числа процессоров, например, в 2 раза задача начинает выполняться больше, чем в 2 раза. Такое возможно, если она начинается помещаться в кэш процессоров, при условии, что раньше не помещалась и приходилось обращаться к медленной памяти.

Рассмотрим 4 основные Графовые модели программ: граф управления программы (ГУ), информационный граф программы (ИГ), операционная история (ОИ), информационная история (ИИ).

Граф управления. Каждому оператору исходной программы поставим в соответствие вершину графа — экземпляр преобразователя или распознавателя в зависимости от типа оператора. Получим множество вершин, между которыми согласно исходной программе определим отношение, соответствующее передаче управления. Если текст программы допускает выполнение одного оператора непосредственно за другим, то соответствующие вершины соединим дугой, направленной от предшественника к последователю. Его основным свойством является

независимость от входных данных программы. Множества вершин и дуг для каждой программы фиксированы и образуют единственный граф.

Информационный граф. Изменим графовую основу. Будем среди операторов принимать во внимание только преобразователи, а в качестве отношения между ними брать отношение информационной зависимости. Построим сначала граф, в котором вершины соответствуют операторам-преобразователям. Две вершины соединим информационной дугой, если между какими-нибудь срабатываниями соответствующих операторов теоретически возможна информационная связь. Информационный граф не зависит от входных данных. В нем могут быть "лишние" дуги, которые не реализуются либо при конкретных входных данных, либо совместно с другими дугами.

Операционная история. Теперь предположим, что каким-то образом мы определили начальные данные программы и наблюдаем за ее выполнением на обычном последовательном вычислителе. Каждое срабатывание каждого оператора (а оно необязательно будет единственным) будем фиксировать отдельной вершиной. Получим множество, которое количественно почти всегда будет отличаться от множества вершин графа управления. В данном случае порядок непосредственного срабатывания операторов можно определить точно. Соединяем вершины дугами передач управления, получаем ориентированный граф. Этот граф представляет единственный путь от начальной вершины к конечной. По существу это не что иное, как последовательность срабатывания преобразователей и распознавателей исходной программы при заданных входных данных. В операционной истории от входных данных зависит практически все: общее число вершин, количество вершин, соответствующих одному оператору, и даже набор присутствующих преобразователей и распознавателей. Для графа управления.

Информационная история. Снова каким-то образом определим начальные данные программы и будем наблюдать за ее выполнением на последовательном вычислителе. Каждое срабатывание каждого оператора-преобразователя будем фиксировать отдельной вершиной. Соединим вершины дугами передач информации, получим ориентированный граф. Для информационного графа. Пусть при фиксированных входных данных программа описывает некоторый алгоритм. Построим ориентированный граф. В качестве вершин возьмем любое множество, например, множество точек арифметического пространства, на которое взаимнооднозначно отображается множество всех операций алгоритма. Возьмем любую пару вершин *u,* v. Допустим, что согласно описанному выше частичному порядку операция, соответствующая вершине *u,* должна поставлять аргумент операции, соответствующей вершине v. Тогда проведем дугу из вершины *u* в вершину *y.* Если соответствующие операции могут выполняться независимо друг от друга, дугу проводить не будем. В случае, когда аргументом операции является начальное данное или результат операции нигде не используется, возможны различные договоренности. Например, можно считать, что соответствующие дуги отсутствуют. Мы будем поступать в зависимости от обстоятельств. Построенный таким образом граф будем называть *графом алгоритма.* Независимо от способа построения ориентированного графа, те его вершины, которые не имеют ни одной входящей или выходящей дуги, будем называть соответственно *входными* или *выходными* вершинами графа. Итак, каждое описание алгоритма порождает ориентированный ациклический мультиграф. Верно и обратное. Если задан ориентированный ациклический мультиграф, то его всегда можно рассматривать как граф некоторого алгоритма. Для этого каждой вершине нужно поставить в соответствие любую однозначную операцию, имеющую столько аргументов, сколько дуг входит в вершину. Поэтому между алгоритмами и рассматриваемыми графами есть определенное взаимное соответствие.

Утверждение 4.1

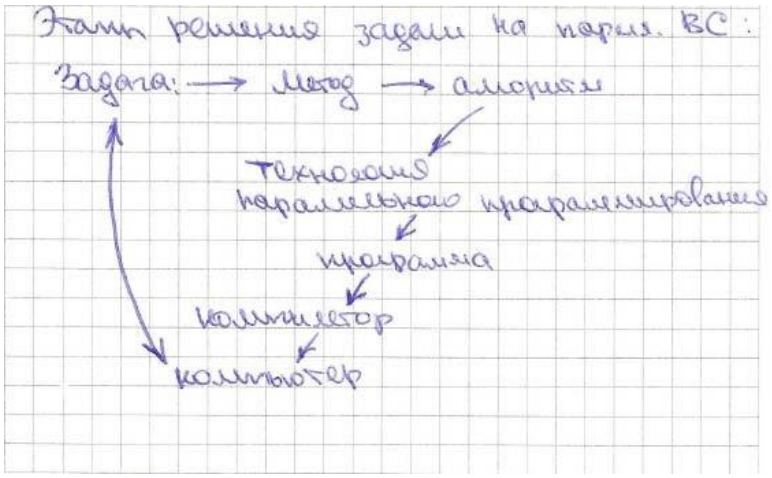
Пусть задан ориентированный ациклический граф, имеющий n вершин. Существует число s < n, для которого все вершины графа можно так пометить одним из индексов 1, 2, ..., s, что если дуга из вершины с индексом i идет в вершину с индексом j, то i<j.

Выберем в графе любое число вершин, не имеющих предшествующих, и пометим их индексом 1. Удалим из графа помеченные вершины и инцидентные им дуги. Оставшийся граф также является ациклическим. Выберем в нем любое число вершин, не имеющих предшествующих, и пометим их индексом 2. Продолжая этот процесс, в конце концов, исчерпаем весь граф. Так как при каждом шаге помечается не менее одной вершины, то число различных индексов не превышает числа вершин графа.

Отсюда следует, что никакие две вершины с одним и тем же индексом не связаны дугой. Минимальное число индексов, которым можно пометить все вершины графа, на 1 больше длины его критического пути. И, наконец, для любого целого числа s, не превосходящего общего числа вершин, но большего длины критического пути, существует такая разметка вершин графа, при которой используются все s индексов.

Граф, размеченный в соответствии с утверждением 4.1, называется **строгой параллельной формой графа**. Если в параллельной форме некоторая вершина помечена индексом k, то это означает, что длины всех путей, оканчивающихся в данной вершине, меньше к. Существует строгая параллельная форма, при которой максимальная из длин путей, оканчивающихся в вершине с индексом k, равна k-1. Для этой параллельной формы число используемых индексов на 1 больше длины критического пути графа. Среди подобных параллельных форм существует такая, в которой все входные вершины

находятся в группе с одним индексом, равным 1. Эта строгая параллельная форма называется **канонической**. Для заданного графа его каноническая параллельная форма единственна. Группа вершин, имеющих одинаковые индексы, называется **ярусом** параллельной формы, а число вершин в группе — шириной яруса. Число ярусов в параллельной форме называется **высотой** параллельной формы, а максимальная ширина ярусов — ее **шириной**. Параллельная форма минимальной высоты называется максимальной.



Итак, вы приступаете к созданию параллельной программы. Желание есть, задача ясна, метод выбран, целевой компьютер, скорее всего, тоже определен. Осталось только все мысли выразить в той или иной форме, понятной для этого компьютера. Чем руководствоваться, если собственного опыта пока мало, а априорной информации о доступных технологиях параллельного программирования явно недостаточно? Наводящих соображений может быть много, но в результате вы все равно будете вынуждены пойти на компромисс, делая выбор между временем разработки программы, ее эффективностью и переносимостью, интенсивностью последующего использования программы, необходимостью ее дальнейшего развития. Не вдаваясь в детали выбора, попробуйте для начала оценить, насколько важны для вас следующие три характеристики.

Основное назначение параллельных компьютеров — это быстро решать задачи. Если технология программирования по разным причинам не позволяет в полной мере использовать весь потенциал вычислительной системы, то нужно ли тратить усилия на ее освоение? Не будем сейчас обсуждать причины. Ясно то, что возможность создания эффективных программ является серьезным аргументом в выборе средств программирования.

Технология может давать пользователю полный контроль над использованием ресурсов вычислительной системы и ходом выполнения его программы. Для этого ему предлагается набор из нескольких сотен конструкций и функций, предназначенных "на все случаи жизни". Он может создать действительно эффективную программу, если правильно воспользуется предложенными средствами. Но захочет ли он это делать? Не стоит забывать, что он должен решать свою задачу из своей предметной области, где и своих проблем хватает. Маловероятно, что физик, химик, геолог или эколог с большой радостью захочет осваивать новую специальность, связанную с параллельным программированием. Возможность быстрого создания параллельных программ должна приниматься в расчет наравне с другими факторами.

Вычислительная техника меняется очень быстро. Предположим, что была найдена технология, позволяющая быстро создавать эффективные параллельные программы. Что произойдет через пару лет, когда появится новое поколение компьютеров? Возможных вариантов развития событий два. Первый вариант — разработанные прежде программы были "одноразовыми" и сейчас ими уже никто не интересуется. Бывает и так. Однако, как правило, в параллельные программы вкладывается слишком много средств (времени, усилий, финансовых затрат), чтобы просто так об этом забыть и начать разработку заново. Хочется перенести накопленный багаж на новую компьютерную платформу. Скорее всего, на новом компьютере старая программа рано или поздно работать будет, и даже будет давать правильный результат. Но дает ли выбранная технология гарантии сохранения эффективности параллельной программы при ее переносе с одного компьютера на другой? Скорее всего, нет. Программу для новой платформы нужно оптимизировать заново. А тут еще разработчики новой платформы предлагают вам очередную новую технологию программирования, которая опять позволит создать выдающуюся программу для данного компьютера. Программы переписываются, и так по кругу в течение многих лет.

Выбор технологии параллельного программирования — это и в самом деле вопрос не простой. Если попытаться сделать обзор средств, которые могут помочь в решении задач на параллельном компьютере, то даже поверхностный анализ приведет к списку из более 100 наименований.

В некоторых случаях выбор определяется просто. Например, вполне жизненной является ситуация, когда воспользоваться можно только тем, что установлено на доступном вам компьютере. Другой аргумент звучит так: "... все используют MPI, поэтому и я тоже буду...". Если есть возможность и желание сделать осознанный выбор, это обязательно нужно делать. Посоветуйтесь со специалистами. Проблемы в дальнейшем возникнут в любом случае, вопрос только насколько

быстро и в каком объеме. Если выбор будет правильным, проблем будет меньше. Если неправильным, то тоже не отчаивайтесь, будет возможность подумать о выборе еще раз. Сделав выбор несколько раз, вы станете специалистом в данной области, забудете о своих прежних интересах, скажем, о квантовой химии или вычислительной гидродинамике. Не исключено, что в итоге вы сможете предложить свою технологию и найти ответ на центральный вопрос параллельных вычислений: "Как создавать эффективные программы для параллельных компьютеров?"

В данной главе мы рассмотрим различные подходы к программированию параллельных компьютеров. Одни широко используются на практике, другие интересны своей идеей, третьи лаконичны и выразительны. Хотелось показать широкий спектр существующих средств, но и не сводить описание каждой технологии до одного абзаца текста. Противоречивая задача. Однако надеемся, что после изучения каждого раздела вы сможете не только проводить качественное сравнение технологий, но и самостоятельно писать содержательные программы. Знакомясь с различными системами параллельного программирования, обязательно обратите внимание на следующее обстоятельство. Если вы решаете учебные задачи или производственные задачи небольшого размера, вам почти наверняка не придется задумываться об эффективности использования параллельной вычислительной техники. В этом случае выбор системы программирования практически не имеет значения. Используйте то, что вам больше нравится. Но как только вы начнете решать большие задачи и, особенно, предельно большие многовариантные задачи, вопрос эффективности может оказаться ключевым.

Очень скоро станет ясно, что при использовании любой системы параллельного программирования желание повысить производительность вычислительной техники на вашей задаче сопровождается тем, что от вас требуется все больше и больше каких-то новых сведений о структуре задачи, программы или алгоритма. Ни одна система параллельного программирования не гарантирует высокую эффективность вычислительных процессов без предоставления дополнительных сведений.

23. Технологии параллельного программирования MPI и OpenMP.

Наиболее распространенной технологией программирования параллельных компьютеров с распределенной памятью в настоящее время является MPI. Мы уже говорили о том, что основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это и отражено в названии данной технологии — Message Passing Interface. Стандарт MPI фиксирует интерфейс, который должны соблюдать как система программирования MPI на каждой вычислительной системе, так и пользователь при создании своих программ. Современные реализации чаще всего соответствуют стандарту MPI версии 1.1. В 1997—1998 годах появился стандарт MPI-2.0, значительно расширивший функциональность предыдущей версии. Однако до сих пор этот вариант MPI не получил широкого распространения. Везде далее, если иного не оговорено, мы будем иметь дело со стандартом 1.1.

MPI поддерживает работу с языками C и Fortran. В данной книге все примеры и описания всех функций будут даны с использованием языка C. Однако это совершенно не является принципиальным, поскольку основные идеи MPI и правила оформления отдельных конструкций для этих языков во многом схожи.

Полная версия интерфейса содержит описание более 120 функций.

Интерфейс поддерживает создание параллельных программ в стиле MIMD, что подразумевает объединение процессов с различными исходными текстами. Однако на практике программисты гораздо чаще используют SPMD-модель, в рамках которой для всех параллельных процессов используется один и тот же код. В настоящее время все больше и больше реализаций MPI поддерживают работу с нитями.

Все дополнительные объекты: имена функций, константы, предопределенные типы данных и т. п., используемые в MPI, имеют префикс MPI_. Например, функция посылки сообщения от одного процесса другому имеет имя MPi_Send. Если пользователь не будет использовать в программе имен с таким префиксом, то конфликтов с объектами MPI заведомо не будет. Все описания интерфейса MPI собраны в файле mpi.h, поэтому в начале MPI-программы должна стоять директива #include <mpi.h>.

MPI-программа — это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается.

Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

Для локализации взаимодействия параллельных процессов программы можно создавать группы процессов, предоставляя им отдельную среду для общения — коммуникатор. Состав образуемых групп произволен. Группы могут полностью входить одна в другую, не пересекаться или пересекаться частично. При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя MPI_COMM_WORLD. Этот коммуникатор существует всегда и служит для взаимодействия всех процессов MPI-программы.

Каждый процесс MPI-программы имеет уникальный атрибут номер процесса, который является целым неотрицательным числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда два основных атрибута процесса: коммуникатор и номер в коммуникаторе.

Если группа содержит n процессов, то номер любого процесса в данной группе лежит в пределах от 0 до n – 1. Подобная линейная нумерация не всегда адекватно отражает логическую взаимосвязь процессов приложения. Например, по смыслу задачи процессы могут располагаться в узлах прямоугольной решетки и взаимодействовать
только со своими непосредственными соседями. Такую ситуацию пользователь может легко отразить в своей программе, описав соответствующую виртуальную топологию процессов. Эта информация может оказаться полезной при отображении процессов программы на физические процессоры вычислительной системы. Сам процесс отображения в MPI никак не специфицируется, однако система поддержки MPI в ряде случаев может значительно уменьшить коммуникационные накладные расходы, если воспользуется знанием виртуальной топологии.

Основным способом общения процессов между собой является посылка сообщений. Сообщение — это набор данных некоторого типа. Каждое сообщение имеет несколько атрибутов, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и др. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от 0 до 32 767. Для работы с атрибутами сообщений введена структура MPistatus, поля которой дают доступ к значениям атрибутов.

На практике сообщение чаще всего является набором однотипных данных, расположенных подряд друг за другом в некотором буфере. Такое сообщение может состоять, например, из двухсот целых чисел, которые пользователь разместил в соответствующем целочисленном векторе. Это типичная ситуация, на нее ориентировано большинство функций MPI, однако такая ситуация имеет, по крайней мере, два ограничения. Во-первых, иногда необходимо составить сообщение из разнотипных данных. Конечно же, можно отдельным сообщением послать количество вещественных чисел, содержащихся в последующем сообщении, но это может быть и неудобно программисту, и не столь эффективно. Во-вторых, не всегда посылаемые данные занимают непрерывную область в памяти. Если в Fortran элементы столбцов матрицы расположены в памяти друг за другом, то элементы строк уже идут с некоторым шагом. Чтобы послать строку, данные нужно сначала упаковать, передать, а затем вновь распаковать.
Чтобы снять указанные ограничения, в MPI предусмотрен механизм для введения производных типов данных (derived datatypes). Описав состав и схему размещения в памяти посылаемых данных, пользователь в дальнейшем работает с такими типами так же, как и со стандартными типами данных MPI.
Поскольку собственные типы данных и виртуальные топологии процессов используются на практике не очень часто, то в данной книге мы не будем их описывать подробно.
MPI: синхронное и асинхронное взаимодействие процессов.
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm)
buf – адрес начала буфера посылки сообщения count – число передаваемых элементов в сообщении datatype – тип передаваемых элементов dest – номер процесса-получателя msgtag – идентификатор сообщения comm – идентификатор группы Блокирующая посылка сообщения с идентификатором msgtag, состоящего из count
элементов типа datatype, процессу с номером dest. Все элементы сообщения расположены подряд в буфере buf. Значение count может быть нулем. Тип передаваемых элементов datatype должен указываться с помощью предопределенных констант типа. Разрешается передавать сообщение самому себе.
Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу dest,
остается за MPI. Следует специально отметить, что возврат из подпрограммы MPI_Send не означает ни того, что сообщение уже передано процессу dest, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший MPI_Send.
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Status *status)
OUT buf – адрес начала буфера приема сообщения count – максимальное число элементов в принимаемом сообщении datatype – тип элементов принимаемого сообщения source – номер процесса-отправителя msgtag – идентификатор принимаемого сообщения comm – идентификатор группы OUT status – параметры принятого сообщения Прием сообщения с идентификатором msgtag от процесса source с блокировкой.
Число элементов в принимаемом сообщении не должно превосходить значения count. Если число принятых элементов меньше значения count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой MPI_Probe.
Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере buf. В качестве номера процесса-отправителя можно указать предопределенную константу MPI_ANY_SOURCE – признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу MPI_ANY_TAG – признак того, что подходит сообщение с любым идентификатором. Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову MPI_Recv, то первым будет принято то сообщение, которое было отправлено раньше.
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count) status – параметры принятого сообщения datatype – тип элементов принятого сообщения OUT count – число элементов сообщения
По значению параметра status данная подпрограмма определяет число уже принятых (после обращения к MPI_Recv) или принимаемых (после обращения к MPI_Probe или MPI_Iprobe) элементов сообщения типа datatype.
int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status) source – номер процесса-отправителя или MPI_ANY_SOURCE msgtag – идентификатор ожидаемого сообщения или MPI_ANY_TAG comm – идентификатор группы
OUT status – параметры обнаруженного сообщения
Получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из подпрограммы не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью параметра status. Следует обратить внимание, что подпрограмма определяет только факт прихода сообщения, но реально его не принимает.

Прием/передача сообщений без блокировки

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)

buf – адрес начала буфера посылки сообщения count – число передаваемых элементов в сообщении datatype – тип передаваемых элементов dest – номер процесса-получателя msgtag – идентификатор сообщения comm – идентификатор группы OUT request – идентификатор асинхронной передачи Передача сообщения, аналогичная MPI_Send, однако возврат из подпрограммы

происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере buf. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации о завершении данной посылки. Окончание процесса передачи (т.е. того момента, когда можно переиспользовать буфер buf без опасения испортить передаваемое сообщение) можно определить с помощью параметра request и процедур MPI_Wait и MPI_Test. Сообщение, отправленное любой из процедур MPI_Send и MPI_Isend, может быть принято любой из процедур MPI_Recv и MPI_Irecv.

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Request *request)

OUT buf – адрес начала буфера приема сообщения count – максимальное число элементов в принимаемом сообщении datatype – тип элементов принимаемого сообщения source – номер процесса-отправителя msgtag – идентификатор принимаемого сообщения comm – идентификатор группы OUT request – идентификатор асинхронного приема сообщения Прием сообщения, аналогичный MPI_Recv, однако возврат из подпрограммы

происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере buf. Окончание процесса приема можно определить с помощью параметра request и процедур MPI_Wait и MPI_Test.

int MPI_Wait(MPI_Request *request, MPI_Status *status) request – идентификатор асинхронного приема или передачи OUT status – параметры сообщения Ожидание завершения асинхронных процедур MPI_Isend или MPI_Irecv,

ассоциированных с идентификатором request. В случае приема, атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра status.

int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses) count – число идентификаторов requests – массив идентификаторов асинхронного приема или передачи OUT statuses – параметры сообщений

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива statuses будет установлено в соответствующее значение.

int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status) count – число идентификаторов, requests – массив идентификаторов асинхронного приема или передачи OUT index – номер завершенной операции обмена OUT status – параметры сообщений Выполнение процесса блокируется до тех пор, пока какая-либо операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если несколько операций могут быть завершены, то случайным образом выбирается одна из них. Параметр index содержит номер элемента в массиве requests, содержащего идентификатор завершенной операции.

int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses) incount – число идентификаторов requests – массив идентификаторов асинхронного приема или передачи. OUT outcount – число идентификаторов завершившихся операций обмена OUT indexes – массив номеров завершившихся операции обмена OUT statuses – параметры завершившихся сообщений. Выполнение процесса блокируется до тех пор, пока по крайней мере одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр outcount содержит число завершенных операций, а первые outcount элементов массива indexes содержат номера элементов массива requests с их идентификаторами. Первые outcount элементов массива statuses содержат параметры завершенных операций.

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status) request – идентификатор асинхронного приема или передачи OUT flag – признак завершенности операции обмена OUT status – параметры сообщения

Проверка завершенности асинхронных процедур MPI_Isend или MPI_Irecv, ассоциированных с идентификатором request. В параметре flag возвращает значение 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра status.

int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses) count – число идентификаторов, requests – массив идентификаторов асинхронного приема или передачи OUT flag – признак завершенности операций обмена OUT statuses – параметры сообщений. В параметре flag возвращает значение 1, если все операции, ассоциированные с указанными идентификаторами, завершены (с указанием параметров сообщений в массиве statuses). В противном случае возвращается 0, а элементы массива statuses неопределены.

int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status) count – число идентификаторов requests – массив идентификаторов асинхронного приема или передачи OUT index – номер завершенной операции обмена
OUT flag – признак завершенности операции обмена OUT status – параметры сообщения Если к моменту вызова подпрограммы хотя бы одна из операций обмена
завершилась, то в параметре flag возвращается значение 1, index содержит номер соответствующего элемента в массиве requests, а status – параметры сообщения.
int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses) incount – число идентификаторов requests – массив идентификаторов асинхронного приема или передачи OUT outcount – число идентификаторов завершившихся операций обмена OUT indexes – массив номеров завершившихся операции обмена OUT statuses – параметры завершившихся операций Данная подпрограмма работает так же, как и MPI_Waitsome, за исключением того, что возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение outcount будет равно нулю.
int MPI_Iprobe(int source, int msgtag, MPI_Comm comm, int *flag, MPI_Status *status) source – номер процесса-отправителя или MPI_ANY_SOURCE msgtag – идентификатор ожидаемого сообщения или MPI_ANY_TAG comm – идентификатор группы
OUT flag – признак завершенности операции обмена OUT status – параметры обнаруженного сообщения Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре flag возвращает значение 1, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично MPI_Probe), и значение 0, если сообщения с указанными атрибутами еще нет.

MPI: различные виды операторов Send.

int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm)
buf – адрес начала буфера посылки сообщения count – число передаваемых элементов в сообщении datatype – тип передаваемых элементов dest – номер процесса-получателя msgtag – идентификатор сообщения comm – идентификатор группы Блокирующая посылка сообщения с идентификатором msgtag, состоящего из count
элементов типа datatype, процессу с номером dest. Все элементы сообщения расположены подряд в буфере buf. Значение count может быть нулем. Тип передаваемых элементов datatype должен указываться с помощью предопределенных констант типа. Разрешается передавать сообщение самому себе.
Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу dest, остается за MPI. Следует специально отметить, что возврат из подпрограммы MPI_Send не означает ни того, что сообщение уже передано процессу dest, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший MPI_Send.
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)
buf – адрес начала буфера посылки сообщения count – число передаваемых элементов в сообщении datatype – тип передаваемых элементов dest – номер процесса-получателя msgtag – идентификатор сообщения, comm – идентификатор группы OUT request – идентификатор асинхронной передачи Передача сообщения, аналогичная MPI_Send, однако возврат из подпрограммы
происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере buf. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации о завершении данной посылки. Окончание процесса передачи (т.е. того момента, когда можно переиспользовать буфер buf без опасения испортить передаваемое сообщение) можно определить с помощью параметра request и процедур MPI_Wait и MPI_Test. Сообщение, отправленное любой из процедур MPI_Send и MPI_Isend, может быть принято любой из процедур MPI_Recv и MPI_Irecv.
int MPI_Send_init(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)
buf – адрес начала буфера посылки сообщения count – число передаваемых элементов в сообщении datatype – тип передаваемых элементов dest – номер процесса-получателя msgtag – идентификатор сообщения comm – идентификатор группы OUT request – идентификатор асинхронной передачи Формирование запроса на выполнение пересылки данных. Все параметры точно
такие же, как и у подпрограммы MPI_Isend, однако в отличие от нее пересылка не начинается до вызова подпрограммы MPI_Startall.
int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype, int dest, int stag, void *rbuf, int rcount, MPI_Datatype rtype, int source, MPI_Datatype rtag, MPI_Comm comm, MPI_Status *status)
sbuf – адрес начала буфера посылки сообщения
scount – число передаваемых элементов в сообщении
stype – тип передаваемых элементов

dest – номер процесса-получателя
stag – идентификатор посылаемого сообщения
OUT *rbuf* – адрес начала буфера приема сообщения
rcount – число принимаемых элементов сообщения
rtype – тип принимаемых элементов
source – номер процесса-отправителя
rtag – идентификатор принимаемого сообщения
comm – идентификатор группы
OUT *status* – параметры принятого сообщения Данная операция объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией *MPI_Sendrecv*, может быть принято обычным образом, и точно также операция *MPI_Sendrecv* может принять сообщение, отправленное обычной операцией *MPI_Send*. Буфера приема и посылки обязательно должны быть различными.

MPI: коллективные операции.

В операциях коллективного взаимодействия процессов участвуют все процессы коммуникатора. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции). int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)
OUT buf – адрес начала буфера посылки сообщения count – число передаваемых элементов в сообщении datatype – тип передаваемых элементов source – номер рассылающего процесса comm – идентификатор группы
Рассылка сообщения от процесса source всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера buf процесса source будет скопировано в локальный буфер процесса. Значения параметров count, datatype и source должны быть одинаковыми у всех процессов.
int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int dest, MPI_Comm comm)
sbuf – адрес начала буфера посылки scount – число элементов в посылаемом сообщении stype – тип элементов отсылаемого сообщения OUT rbuf – адрес начала буфера сборки данных rcount – число элементов в принимаемом сообщении rtype – тип элементов принимаемого сообщения dest – номер процесса, на котором происходит сборка данных comm – идентификатор группы OUT ierror – код ошибки Сборка данных со всех процессов в буфере rbuf процесса dest. Каждый процесс,
включая dest, посылает содержимое своего буфера sbuf процессу dest. Собирающий процесс сохраняет данные в буфере rbuf, располагая их в порядке возрастания номеров процессов. Параметр rbuf имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров count, datatype и dest должны быть одинаковыми у всех процессов.
int MPI_Allreduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
sbuf – адрес начала буфера для аргументов OUT rbuf – адрес начала буфера для результата count – число аргументов у каждого процесса datatype – тип аргументов op – идентификатор глобальной операции comm – идентификатор группы Выполнение count глобальных операций op с возвратом count результатов во всех
процессах в буфере rbuf. Операция выполняется независимо над соответствующими аргументами всех процессов. Значения параметров count и datatype у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция op обладает свойствами ассоциативности и коммутативности.
int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
sbuf – адрес начала буфера для аргументов OUT rbuf – адрес начала буфера для результата count – число аргументов у каждого процесса datatype – тип аргументов op – идентификатор глобальной операции root – процесс-получатель результата comm – идентификатор группы Функция аналогична предыдущей, но результат будет записан в буфер rbuf только у процесса root.
int MPI_Barrier(MPI_Comm comm) comm – идентификатор группы Блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы comm также не выполнят эту процедуру.

MPI: группы, коммуникаторы.

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
comm – идентификатор группы
color – признак разделения на группы
key – параметр, определяющий нумерацию в новых группах

OUT *newcomm* – идентификатор новой группы Данная процедура разбивает все множество процессов, входящих в группу *comm*, на непересекающиеся подгруппы – одну подгруппу на каждое значение параметра *color* (неотрицательное число). Каждая новая подгруппа содержит все процессы одного цвета. Если в качестве *color* указано значение *MPI_UNDEFINED*, то в *newcomm* будет возвращено значение *MPI_COMM_NULL*. int MPI_Comm_free(MPI_Comm comm)

• OUT *comm* – идентификатор группы

Уничтожает группу, ассоциированную с идентификатором *comm*, который после возвращения устанавливается в *MPI_COMM_NULL*.