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

Определение 1. Пусть A $\subset$ P^2. Тогда замыканием A называется множество всех функций алгебры логики, которые можно выразить формулами над A. Замыкание обозначается как [A].
Свойства замыкания:

1) [A] $\supseteq$ A;
2) A $\supseteq$ B ==> [A] $\supseteq$ [B], причём, если в левой части импликации строгое вложение, то из него вовсе не следует строгое вложение в правой части — верно лишь A $\supset$ B ==> [A] $\supseteq$ [B];
3) [[A]] = [A].

Определение 2. Система функций алгебры логики A называется полной, если [A] = P^2.

Определение 3. Пусть A $\subset$ P^2. Тогда система A называется замкнутым классом, если замыкание A совпадает с самим A: [A] = A.

Утверждение. Пусть A — замкнутый класс, A $\neq$ P^2 и B $\subset$ A. Тогда B — неполная система (подмножество неполной системы будет также неполной системой).
Доказательство. B $\subset$ A ==> [B] $\subset$ [A] = A $\neq$ P2 ==> [B] $\neq$ P2. Следовательно, B — неполная система. Утверждение доказано.

Теорема. Класс $T_0 = \{f(x_1, \ldots, x_n) \mid f(0, \ldots, 0) = 0\}$ —замкнутый.
Доказательство. Пусть $\{f(x_1,..,x_n), g_1(y_{11},...y_{1,m1}),\ldots,g_n(y_{n1},\ldots,y_{n,mn})\} \subset T_0$. Рассмотрим функцию $h(y_1,\ldots,y_r) = f(g_1(y_{11},\ldots,y_{1,m1}),\ldots,g_n(y_{n1},\ldots,y_{m,nn}))$. Среди переменных функций g_i могут встречаться и одинаковые, поэтому в качестве переменных функции h возьмём все различные из них. Тогда $h(0, \ldots, 0) = f(g_1(0, \ldots, 0), \ldots, g_n(0, \ldots, 0)) = f(0, \ldots, 0) = 0$, следовательно, функция h также сохраняет ноль. Рассмотрен только частный случай (без переменных в качестве аргументов). Однако, поскольку тождественная функция сохраняет ноль, подстановка простых переменных эквивалентна подстановке тождественной функции, теорема доказана.

Теорема. Класс $T_1 = \{f(x_1, \ldots, x_n) \mid f(1, 1, \ldots, 1) = 1\}$ замкнут.
Доказательство повторяет доказательство аналогичной теоремы для класса T_0.

Определение. Функция алгебры логики $f(x_1, \ldots, x_n)$ называется линейной, если $f(x_1, \ldots, x_n) = a_0 \oplus a_1x_1 \oplus \ldots \oplus a_nx_n$, где $a_i \in \{0, 1\}$. Иными словами, в полиноме линейной функции нет слагаемых, содержащих конъюнкцию.

Теорема. Класс L замкнут.
Доказательство. Поскольку тождественная функция — линейная, достаточно рассмотреть только случай подстановки в формулы функций: пусть $f(x_1, \ldots, x_n) \in L$ и $g_i \in L$. Достаточно доказать, что $f(g_1, \ldots, g_n) \in L$. Действительно, если не учитывать слагаемых с коэффициентами $a_i = 0$, то всякую линейную функцию можно представить в виде $x_{i1} \oplus x_{i2} \oplus x_{ik} \oplus a_0$. Если теперь вместо каждого x_{ij} подставить линейное выражение,
то получится снова линейное выражение (или константа единица или нуль).

Определение. Функцией, двойственной к функции алгебры логики $f(x_1, \ldots, x_n)$, называется функция $f^*(x_1,\ldots,x_n) = \overline{f}(\overline{x_1},\ldots,\overline{x_n})$
Теорема (принцип двойственности). Пусть $\Phi(y_1,\ldots,y_m) = f(g_1(y_{11},\ldots,y_{1k1}),\ldots,(y_{n1},\ldots,y_{nkn}))$. Тогда $\Phi^*(y_1,\ldots,y_m) = f^*(g_1^*(y_{11}, \ldots,y_{1k1}),\ldots,g_n^*(y_{n1},\ldots,y_{nkn}))$.
Доказательство. Рассмотрим $\Phi^*(y_1,\ldots,y_m) = \overline{f}(g_1(\overline{y_{11}},\ldots,\overline{y_{1k1}}),\ldots,g_n(\overline{y_{n1}},\ldots,\overline{y_{nkn}})) = \overline{f}(\overline{\overline{g}_{11}},\ldots,\overline{y_{1k1}}),\ldots,\overline{\overline{g}_n(\overline{y_{n1}},\ldots,\overline{y_{nkn}})} = \overline{f}(\overline{g_1^*}(y_{11},\ldots,y_{1k1}),\ldots,\overline{g_n^*}(y_{n1},\ldots,y_{nkn})) = f^*(g_1^*(y_{11},\ldots,y_{1k1}),\ldots,g_n^*(y_{n1},\ldots,y_{nkn}))$.
Следствие. Пусть функция $\Phi(y_1, \ldots, y_m)$ реализуется формулой над A = {f1, f2, …}. Тогда если в этой формуле всюду заменить вхождения f_i на f_i^*, то получится формула, реализующая $\Phi^*(y_1, \ldots, y_m)$.
Утверждение. Для любой функции алгебры логики $f(x_1, \ldots, x_n)$ справедливо равенство $f(x_1, \ldots, x_n)=f^{**}(x_1, \ldots, x_n)$.

Определение. Функция алгебры логики $f(x_1, \ldots, x_n)$ называется самодвойственной, если $f(x_1, \ldots, x_n) = f^*(x_1, \ldots, x_n)$. Иначе говоря, $S = \{f \mid f = f^*\}$.
Теорема. Класс S замкнут.
Доказательство. Пусть $f(x_1, \ldots, x_n) \in S$, для любого i $g_i(y_{i1},\ldots,y_{iki}) \in S$, $i = 1, 2, \ldots, n$ и $\Phi = f(g_1(y_{11},\ldots,y_{1k1}),\ldots,g_n(y_{n1},\ldots,y_{nkn}))$. Тогда из принципа двойственности следует, что $\Phi^* = f^*(g_1^*(y_{11},\ldots,y_{1k1}),\ldots,g_n^*(y_{n1},\ldots,y_{nk}))$. Таким образом, мы получили, что $\Phi = \Phi^*$ и $\Phi \in S$.

Определение. Функция алгебры логики $f(x_1, \ldots, x_n)$ называется монотонной, если для любых двух сравнимых наборов α и β выполняется импликация α <= β ==> f(α) <= f(β) .

Теорема. Класс M монотонных функций замкнут.
Доказательство. Поскольку тождественная функция монотонна, достаточно проверить лишь случай суперпозиции функций. Пусть $f(x_1, \ldots, x_n) \in M$, для любого j $g_j(y_1, \ldots, y_m) \in M$ и $\Phi(y_1, \ldots, y_m) = f(g_1(y_1, \ldots, y_m), \ldots, g_n(y_1, \ldots, y_m))$. Рассмотрим произвольные наборы $\alpha = (\alpha_1, \ldots, \alpha_m)$ $\beta = (\beta_1, \ldots \beta_m)$ такие, что $\alpha <= \beta$. Обозначим $g_i(\alpha) = \gamma_i$, $g_i(\beta) = \delta_i$. Тогда для любого i имеем $\gamma_i <= \delta_i$. Тогда по определению $\gamma = (\gamma_1, \ldots, \gamma_m) <= \delta = (\delta_1, \ldots, \delta_m)$ и, в силу монотонности функции f, $f(\gamma) <= f(\delta)$. Но $\Phi(\alpha) = f(\gamma_1, \ldots, \gamma_m) = f(\gamma)$, $\Phi(\beta) = f(\delta_1, \ldots, \delta_m) = f(\delta)$ и неравенство $f(\gamma) <= f(\delta) <==> \Phi(\alpha) <= \Phi(\beta)$, следовательно, $\Phi \in M$.

Лемма (о несамодвойственной функции). Из любой несамодвойственной функции алгебры логики $f(x_1, \ldots, x_n)$, подставляя вместо всех переменных функции $\overline{x}$ и x, можно получить $\varphi(x) \equiv const$.
Доказательство. Пусть $f \notin S$. Тогда $\overline{f}(\overline{x_1}, \ldots, \overline{x_n}) \neq f(x_1, \ldots x_n) ==> \exists \sigma = (\sigma_1, \ldots, \sigma_n)$: $\overline{f}(\overline{\sigma_1}, \ldots, \overline{\sigma_n}) \neq f(\sigma_1, \ldots \sigma_n) <==> f(\overline{\sigma_1}, \ldots, \overline{\sigma_n}) = f(\sigma_1, \ldots, \sigma_n)$. Построим функцию $\varphi(x)$ так: $\varphi(x) = f(x \oplus \sigma_1, \ldots, x \oplus \sigma_n)$. Действительно, $\varphi(0) = f(\sigma_1, \ldots, \sigma_n)$, $\varphi(1) = f(\overline{\sigma_1}, \ldots, \overline{\sigma_n})$ и $\varphi(0) = \varphi(1) ==> \varphi(x) = const$. Заметим, что подстановка удовлетворяет условию теоремы, так как

$$x \oplus \sigma = \begin{cases} x, \sigma = 0 \\ \overline{x}, \sigma = 1 \end{cases}$$

Лемма (о немонотонной функции). Из любой немонотонной функции алгебры логики $f(x_1, \ldots, x_n)$, подставляя вместо всех переменных функции x, 0, 1, можно получить функцию $\varphi(x) \equiv \overline{x}$.
Доказательство. Пусть $f \notin M$. Тогда существуют такие наборы $\alpha = (\alpha_1, \ldots, \alpha_n)$ и $\beta = (\beta_1, \ldots, \beta_n)$, что $\alpha < \beta$ и $f(\alpha) > f(\beta)$. Выделим те разряды $i_1, \ldots, i_k$ наборов α и β, в которых они различаются. Очевидно, в наборе α эти разряды равны 0, а в наборе β — 1. Рассмотрим последовательность наборов $\alpha^0, \alpha^1, \ldots, \alpha^k$, таких, что $\alpha = \alpha^0 < \alpha^1 < \alpha^2 < \ldots < \alpha^k = \beta$, где α^{i+1} получается из α^i заменой одного из нулей на 1. Поскольку $f(\alpha) = 1$, а $f(\beta) = 0$, то найдутся соседние α^i и α^{i+1}, что $f(\alpha^i) = 1$ и $f(\alpha^{i+1}) = 0$. Пусть они различаются в r-ом разряде: $\alpha^i = (\alpha_1, \alpha_2, \ldots, \alpha_{r-1}, 0, \alpha_{r+1}, \ldots, \alpha_n)$, $\alpha^{i+1} = (\alpha_1, \alpha_2, \ldots, \alpha_{r-1}, 1, \alpha_{r+1}, \ldots, \alpha_n)$.
Тогда определим функцию $\varphi(x)$ так: $\varphi(x) = f(\alpha_1, \alpha_2, \ldots, \alpha_{r-1}, x, \alpha_{r+1}, \ldots, \alpha_n)$. Действительно, тогда $\varphi(0) = f(\alpha^i) = 1$, $\varphi(1) = f(\alpha^{i+1}) = 0$, $\varphi(x) \equiv \overline{x}$.

Определение. Полиномом Жегалкина над $x_1, \ldots, x_n$ называется выражение вида $K_1 \oplus K_2 \oplus K_3 \oplus \ldots \oplus K_l$, либо 0, где $l >= 1$ и все K_j есть выражения вида $x_{i1}x_{i2}\ldots x_{ik}$, где все переменные различны, либо 1.

Теорема(теорема Жегалкина). Любую функцию алгебры логики $f(x_1, \ldots, x_n)$ можно единственным образом выразить полиномом Жегалкина над $x_1, \ldots, x_n$.

Лемма (о нелинейной функции). Из любой нелинейной функции алгебры логики $f(x_1, \ldots, x_n)$, подставляя вместо всех переменных x, $\overline{x}$, y, $\overline{y}$, 0, 1, можно получить $\varphi(x, y) = xy$ или $\varphi(x, y) = \overline{xy}$.
Доказательство. Пусть $f(x_1, \ldots, x_n) \notin L$. Рассмотрим полином Жегалкина этой функции. Из её нелинейности следует, что в нём присутствуют слагаемые вида $x_{i1}x_{i2}$. Не ограничивая общности рассуждений, будем считать, что присутствует произведение $x_1x_2 \cdot \ldots$. Таким образом, полином Жегалкина этой функции выглядит так:
$f(x_1, \ldots, x_n) = x_1 \cdot x_2 \cdot P_1(x_3, \ldots, x_n) \oplus x_1 \cdot P_2(x_3, \ldots, x_n) \oplus x_2 \cdot P_3(x_3, \ldots, x_n) \oplus P_4(x_3, \ldots, x_n)$,
причем $P_1(x_3, \ldots, x_n) \neq 0$. Иначе говоря, $\exists a_3, a_4, \ldots, a_n \in E2 = \{0, 1\}$ такие, что $P_1(a_3, a_4, \ldots, a_n) = 1$. Рассмотрим вспомогательную функцию $f(x_1, x_2, a_3, a_4, \ldots, a_n) = x_1x_2 \cdot {}_1 \oplus x_1 \cdot b \oplus x2 \cdot c \oplus d$. Тогда функция $f(x \oplus c, y \oplus b, a_3, a_4, \ldots, a_n) = (x \oplus c)(y \oplus b) \oplus (x \oplus c)b \oplus (y \oplus b)c \oplus d = xy \oplus x \cdot b \oplus y \cdot c \oplus b \cdot c \oplus x \cdot b \oplus b \cdot c \oplus y \cdot c \oplus b \cdot c \oplus d = xy \oplus (bc \oplus d) =$

$$\begin{cases} xy, bc \oplus d = 0 \\ \overline{xy}, bc \oplus d = 1 \end{cases}$$

Теорема 12 (теорема Поста). Система функций алгебры логики $A = \{f_1, f_2, \ldots\}$ 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 T0, T1, S, L, M и пусть $A \subset N$. Тогда $[A] \subset [N] = N \neq P2$ и $[A] \neq P2$.
Достаточность. Пусть $A \not\subset T0$, $A \not\subset T1$, $A \not\subset M$, $A \not\subset L$, $A \not\subset S$. 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 $\overline{x}$. Рассмотрим функцию $f_0(x_1, \ldots, x_n) \notin T_0$ и введём функцию $\varphi_0(x) = f_0(x, x, \ldots, x)$. Так как функция f_0 не сохраняет нуль, $\varphi_0(0) = 1$. Возможны два случая: либо $\varphi_0(x) = \overline{x}$, либо $\varphi_0(x) \equiv 1$. Рассмотрим функцию $f_1(x_1, \ldots, x_n) \notin T_1$ и аналогичным образом введём функцию $\varphi_1(x) = f_1(x, x, \ldots, x)$. Так как функция f_1 не сохраняет единицу, $\varphi_1(1) = 0$. Возможны также два случая: либо $\varphi_1(x) = \overline{x}$, либо $\varphi_1(x) \equiv 0$. Если хотя бы в одном случае получилось искомое отрицание, пункт завершён. 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.
b) Получение констант 0 и 1. Имеем $f_S \notin S$. Согласно лемме о несамодвойственной

функции, подставляя вместо всех переменных функции fS отрицание (которое получено в пункте а) и тождественную функцию, можно получить константы:
$[f_S, \overline{x}\,] \supseteq [0, 1]$.
с) Получение конъюнкции $x \cdot y$. 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:
В результате получено, что $[f_0, f_1, f_M, f_L, f_S] \supseteq [\overline{x}, xy] = P^2$.

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

Определение. Графом называется произвольное множество элементов V и произвольное семейство E пар из V.
Обозначение: G = (V, E).
В дальнейшем будем рассматривать конечные графы, то есть графы с конечным множеством элементов и конечным семейством пар.
Определение. 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. Пара вида (a, a) называется петлёй, если пара (a, b) встречается в семействе E несколько раз, то она называется кратным ребром (кратной дугой).
Определение. 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. Две вершины графа называются смежными, если они соединены ребром.
Определение. Говорят, что вершина и ребро инцидентны, если ребро содержит вершину.
Определение. Степенью вершины (deg v) называется количество рёбер, инцидентных данной вершине. Для псевдографа полагают учитывать петлю дважды.

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

$\sum_{i=1}^{p} \deg v_i = 2q$, где p – число вершин, а q – число ребер.

Определение. Пусть множество вершин графа $V = \{v_1, v_2, \ldots, v_p\}$. Тогда матрицей смежности этого графа назовём матрицу $A = ||a_{ij}||$, где $a_{ij} = 1$, если вершины v_i и v_j смежны
(2, 3, … для мультиграфа или псевдографа) и 0 в противном случае, a_{ii} при этом равно числу петель в вершине v_i.
Определение. Два графа (или псевдографа) $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ называются изоморфными, если существуют два взаимно однозначных отображения φ_1: $V_1 \to V_2$ и
φ_2: $E_1 \to E_2$ такие, что для любых двух вершин u и v графа G_1 справедливо $\varphi_2(u, v) = (\varphi_1(u), \varphi_1(v))$.
Определение (изоморфизм графов без петель и кратных рёбер). Два графа $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ называются изоморфными, если существует взаимно однозначное отображение φ_1: $V_1 \to V_2$ такое, что $(u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$.
Определение. Граф $G_1 = (V_1, E_1)$ называется подграфом графа G = (V, E), если $V_1 \subset V$, $E_1 \subset E$.
Определение. Путём в графе G = (V, E) называется любая последовательность вида $v_0, (v_0, v_1), v_1, (v_1, v_2), \ldots, v_{n-1}, (v_{n-1}, v_n), v_n$.
Число n в данных обозначениях называется длиной пути.
Определение. Цепью называется путь, в котором нет повторяющихся рёбер. Простой цепью называется путь без повторения вершин.
Утверждение. Пусть в G = (V, E) $v_1 \neq v_2$ и пусть P — путь из v_1 в v_2. Тогда в P можно выделить подпуть из v_1 в v_2, являющийся простой цепью.
Доказательство. Пусть данный путь — не простая цепь. Тогда в нём повторяется некоторая вершина v, то есть он имеет вид: $P_1 = v_0C_1vC_2vC_3v_2$. Тогда он содержит подпуть $P_2 = v_0C_1vC_3v_2$. Если в P_2 повторяется некоторая вершина, то аналогично удалим ещё кусок и так далее. Процесс должен закончиться, так как P_1 — конечный путь. **Определение**. Путь называется замкнутым, если $v_0 = v_n$. Путь называется циклом, если он замкнут, и рёбра в нём не повторяются. Путь называется простым циклом, если $v_0 = v_n$ и вершины не повторяются.
Определение. Граф G = (V, E) называется связным, если для любых вершин $v_i, v_j \in V$ ($v_i \neq v_j$) существует путь из v_i в v_j.

Рассмотрим отношение $v_i \to v_j$ существования пути из v_i в v_j для неориентированных графов. Оно
1) симметрично, так как $(v_i \to v_j) \Longrightarrow (v_j \to v_i)$,
2) транзитивно, так как $(v_i \to v_j)$ & $(v_j \to v_k) \Longrightarrow (v_i \to v_k)$,
3) рефлексивно, так как для любого i $(v_i \to v_i)$.
Таким образом, получено, что $v_i \to v_j$ — отношение эквивалентности на множестве неориентированных графов и множество вершин разбивается на конечное число классов эквивалентности: $V \to V_1 \cup V_2 \cup \ldots \cup V_k$, $Vi \cap Vj = \varnothing$ если $i \neq j$. При этом граф G разбивается на связные подграфы, которые называются компонентами связности.

Определение. Деревом называется связный граф без циклов.

Определение. 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 $V_1 = V$.
Определение 2. 1) Граф, состоящий из одной вершины, которая выделена, называется корневым деревом.
2) Пусть имеются корневые деревья $D_1, D_2, \ldots, D_m$ с корнями $v_1, v_2, \ldots, v_m$, $D_i = (V_i, E_i)$, $V_i \cap V_j = \varnothing$ $(i \neq j)$. Тогда граф $D = (V, E)$, полученный следующим образом:
$V = V_1 \cup V_2 \cup \ldots \cup V_m \cup \{v\}$ $(v \notin V_i,$ для любого $i)$, $E = E_1 \cup E_2 \cup \ldots \cup E_m \cup \{(v, v_1), (v, v_2), \ldots, (v, v_m)\}$ 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. Упорядоченным корневым деревом называется корневое дерево, в котором
1) задан порядок поддеревьев и
2) каждое поддерево D_i является упорядоченным поддеревом.

Теорема. 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 $2^{2q} = 4^q$.

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.

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

Теорема (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 $q >= p - 1$.
a) Базис индукции: $q = p - 1$. Так как G — связный и $q = p - 1$, то G — дерево, то есть, в G нет циклов. Тогда $r = 1$. Отсюда $p - q + r = 2$.
b) Пусть для q: $p - 1 <= q < q_0$ 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 в G есть цикл (т.к. добавление любого ребра к связному графу порождает 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 $p - q_0 + r = 2$.

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

Определение. Алфавитом называется множество $V \cup C \cup F \cup P \cup L \cup Q \cup S$, где

1) $V = \{x_1, \ldots, x_n\}$ – предметные переменные
2) $C = \{c_1, \ldots, c_m\}$ – предметные константы
3) $F = \{f_1, \ldots, f_k\}$ – функциональные символы. Каждый функциональный символ снабжен арностью этого символа.
4) $P = \{P_1, \ldots, P_l\}$ – предикатные символы (также с арностями)
5) $L = \{ \neg , \&, \vee , \rightarrow \}$ – логические связки
6) $Q = \{ \exists , \forall \}$ – кванторы
7) S – скобки и знаки препинания.

Определение. Термом называется конструкция, построенная по следующим правилам:

1) $x \in V \cup C$ – терм

2) f$\in$ F и имеет арность n, $t_1,...t_n$ – термы. Тогда $f(t_1,...,t_n)$ – терм.
3) Других термов нет.

Определение. Формулой называется конструкция, построенная по следующим правилам:

1) $P_i\in$ P и имеет арность n, $t_1,..., t_n$ – термы, тогда $P_i(t_1,..., t_n)$ – атомарная формула (атом).
2) ϕ , φ - формулы, тогда ($\phi \vee \varphi$), ($\phi \& \varphi$), ($\neg \phi$), ($\phi \to \varphi$) – формулы
3) ϕ - формула, $x\in$ V, тогда ($\exists x \phi$), ($\forall x \phi$) – формулы
4) Других формул нет.

Определение. Пусть Q квантор; подформула ϕ формулы (Qx ϕ) называется областью действия квантора Q по переменной x. Переменная x называется свободной в формуле φ , если не входит в область действия никакого квантора по x. В формуле (Qx ϕ) квантор Q связывает все свободные вхождения переменной x в формулу ϕ . Формула называется замкнутой, если содержит только связанные вхождения переменных.

Определение. Интерпретацией I = (D_I, $\overline{C}$, $\overline{F}$, $\overline{P}$), где

1) D_I –непустое множество, область интерпретации
2) $\overline{C}$:C $\to$ D_I
3) $\overline{F}$:F $\to$ ($D^* \to D_I$), где D^* - множество подмножеств предметов из D_I, причем отображение $\overline{F}$ сохраняет арность.
4) $\overline{P}$:P $\to$ ($D^* \to$ {*истина*, *ложь*}), сохраняющее арность

Определение. Пусть $t(x_1,...,x_n)$ – терм, I – интерпретация, $b_1,...,b_n \in D_I$. Тогда значением терма на в I на наборе $b_1,...,b_n$ называется конструкция $t(x_1,...,x_n)[b_1,...,b_n]_I$, построенная по следующим правилам:

1) Если t = c$\in$ C, то $t(x_1,...,x_n)[b_1,...,b_n]_I = \overline{C}$ (c) $\in D_I$
2) Если t = $x_i\in$ V, то $t(x_1,...,x_n)[b_1,...,b_n]_I = b_i$
3) Если t = f($t_1,...,t_m$)$\in$ F, то $t(x_1,...,x_n)[b_1,...,b_n]_I = \overline{F}$ (f)($t_1(x_1,...,x_n)[b_1,...,b_n]_I$,..., $t_m(x_1,...,x_n)[b_1,...,b_n]_I$)

Далее будем использовать краткую запись: $t(x_1,...,x_n)[b_1,...,b_n]_I \sim t[b_1,...,b_n]$

Определение. Пусть ϕ ($x_1,...,x_n$) - формула, I – интерпретация, $b_1,...,b_n\in D_I$. Будем говорить, что ϕ выполнима в I на наборе $b_1,...,b_n$, (I |= ϕ ($x_1,...,x_n$)[$b_1,...,b_n$]) если выполнены следующие условия:

1) Если $\phi = P_i(t_1(x_1,...,x_n),...,t_m(x_1,...,x_n))$, то I |= ϕ ($x_1,...,x_n$)[$b_1,...,b_n$] $\Leftrightarrow$ $P_i(t_1(b_1,...,b_n),...,t_m(b_1,...,b_n))$ = *истина*.
2) Если
   a) $\phi (x_1,...,x_n) = \neg \phi_1(x_1,...,x_n)$
   b) $\phi (x_1,...,x_n) = \phi_1(x_1,...,x_n) \& \phi_2(x_1,...,x_n)$
   c) $\phi (x_1,...,x_n) = \phi_1(x_1,...,x_n) \vee \phi_2(x_1,...,x_n)$
   d) $\phi (x_1,...,x_n) = \phi_1(x1,...,x_n) \to \& \phi_2(x1,...,x_n)$

   то I |= ϕ ($x_1,...,x_n$)[$b_1,...,b_n$] $\Leftrightarrow$
   e) не верно, что I |= $\phi_1(x_1,...,x_n)$[$b_1,...,b_n$]
   f) одновременно I |= $\phi_1(x_1,...,x_n)$[$b_1,...,b_n$] и I |= $\phi_2(x_1,...,x_n)$[$b_1,...,b_n$]
   g) I |= $\phi_1(x_1,...,x_n)$[$b_1,...,b_n$] или I |= $\phi_2(x_1,...,x_n)$[$b_1,...,b_n$]
   h) I |= $\phi_2(x_1,...,x_n)$[$b_1,...,b_n$] или не верно, что I |= $\phi_1(x_1,...,x_n)$[$b_1,...,b_n$]
3) Если $\phi (x_1,...,x_n) = (\exists [\forall] x_0 \varphi (x_0, x_1,...,x_n))$, то I |= ϕ ($x_1,...,x_n$)[$b_1,...,b_n$] $\Leftrightarrow$ $\exists [\forall] b_0\in D_I$, I |= φ ($x_0, x_1,...,x_n$)[$b_0, b_1,...,b_n$]

Далее будем использовать краткую запись: I |= ϕ ($x_1,...,x_n$)[$b_1,...,b_n$] ~ I |= ϕ [$b_1,...,b_n$]

Определение.

1) ϕ ($x_1,...,x_n$) выполнима в I (I |= ϕ) если $\exists$ $b_1,...,b_n$, что I |= ϕ [$b_1,...,b_n$]
2) ϕ ($x_1,...,x_n$) истинна в I (I |= ϕ) если $\forall$ $b_1,...,b_n$, I |= ϕ [$b_1,...,b_n$]
3) ϕ ($x_1,...,x_n$) выполнима $\Leftrightarrow$ $\exists$ I, что I |= ϕ
4) ϕ ($x_1,...,x_n$) противоречива $\Leftrightarrow$ $\forall$ I, ϕ не выполнима в I
5) ϕ ($x_1,...,x_n$) общезначима (|= ϕ) $\Leftrightarrow$ $\forall$ I, I |= ϕ

Определение. Пусть $\Gamma = \{\phi_1, \phi_2, ...\}$ – конечное или бесконечное множество замкнутых формул. I – модель для Г, если $\forall \phi_i$ I |= ϕ_i.

Определение. Пусть ϕ_0 – замкнута. ϕ_0 называется логическим следованием $\Gamma = \{\phi_1, \phi_2, ...\}$, если для любой модели I для Г, I |= ϕ_0 (обозначается Г |= ϕ_0).

Теорема (о логическом следовании). Пусть $\Gamma = \{\phi_1,..., \phi_n\}$ – конечное множество формул, ϕ_0 – замкнутая формула. Г |= ϕ_0 т. и т.т., когда |= ($\phi_1 \& ... \& \phi_n) \to \phi_0$.

Доказательство. Необходимость. Г |= ϕ_0. Рассмотрим произвольную интерпретацию I.

Если I – модель для Г, то I |= ϕ_0, и I |= ($\phi_1 \& ... \& \phi_n) \to \phi_0$.

Если же I – не модель для Г, до $\exists\ \Phi_i$, что Φ_i не выполнима на I, следовательно, $\Phi_1\&...\&\Phi_n$ не выполнима на I и I |= ($\Phi_1\&...\&\Phi_n$)→ Φ_0.
Достаточность. |= ($\Phi_1\&...\&\Phi_n$)→ Φ_0. Пусть I – произвольная модель для Г, тогда $\forall\ \Phi_i$, I |= Φ_i. Значит, I |= $\Phi_1\&...\&\Phi_n$. Т.к. I |= ($\Phi_1\&...\&\Phi_n$)→ Φ_0 (следует из общезначимости ($\Phi_1\&...\&\Phi_n$)→ Φ_0), то необходимо I |= Φ_0, а значит и Г |= Φ_0.

Определение. Пусть Φ и φ - формулы. Φ и φ равносильны ($\Phi \equiv \varphi$), если ($\Phi \to \varphi$)&($\varphi \to \Phi$).
Утверждение. Если |= Φ и |= $\Phi \equiv \varphi$, то |= φ. Если Φ - невыполнима, и |= $\Phi \equiv \varphi$, то и φ - невыполнима.

Теорема. Следующие пары формул эквивалентны:

1) ($\Phi \& \varphi$) и ($\varphi \& \Phi$)
2) ($\Phi \& \varphi$)$\& \chi$ и $\varphi \&$($\Phi \& \chi$)
3) ($\Phi \vee \varphi$) и ($\varphi \vee \Phi$)
4) ($\Phi \vee \varphi$)$\vee \chi$ и $\varphi \vee$ ($\Phi \vee \chi$)
5) ($\Phi \to \varphi$) и ($\neg \Phi \vee \varphi$)
6) $\neg$ ($\Phi \& \varphi$) и ($\neg \varphi$)$\vee$ ($\neg \Phi$)
7) $\neg$ ($\Phi \vee \varphi$) и ($\neg \varphi$)&($\neg \Phi$)
8) $\neg$ ($\neg \Phi$) и Φ
9) $\neg$ ($\forall x\ \Phi$) и $\exists x \neg \Phi$
10) $\neg$ ($\exists x\ \Phi$) и $\forall x \neg \Phi$
11) $Qx\ \Phi$ и $Qy\ \Phi\{x/y\}$, где Q – квантор и y не содержится в $Qx\ \Phi$
12) ($Qx\ \Phi$) &[$\vee$] φ и Qx (Φ &[$\vee$]φ), x не содержится в φ свободно
13) ($\Phi \vee \varphi$)$\& \chi$ и ($\varphi \& \chi$)$\vee$ ($\Phi \& \chi$)
14) $\Phi \& \Phi$ и Φ
15) $\Phi \vee \Phi$ и Φ и т.д.

Теорема (о замене эквивалентных подформул). Если формулы Φ' и φ' равносильны, и формула φ получается из Φ заменой подформулы Φ' на подформулу φ', то формулы Φ и φ равносильны.

Определение. Замкнутая формула Φ имеет предваренную нормальную форму, если $\Phi = Q_1x_1...Q_nx_nM(x_1,...,x_n)$, где Qi – кванторы и M не содержит кванторов и представлена в КНФ, т.е. $M = K_1\& ...\&K_m$, $K_i = L_1 \vee ... \vee L_k$, L_i – либо атом, либо отрицание атома.

Теорема. Для любой замкнутой формулы Φ существует предваренная нормальная форма.

Определение. Пусть $\Phi = Q_1x_1...Q_nx_nM(x_1,...,x_n)$ - формула в п.н.ф. Тогда формула Φ' называется скулемовской нормальной формой (с.н.ф.) формулы Φ, если она получена из Φ следующими преобразованиями: если Q_i - квантор $\exists$, а $Q_1...Q_{i-1}$ - все кванторы $\forall$, предшествующие Q_i в Φ, то Q_ix_i удаляется из кванторной приставки, а в формуле M каждое вхождение переменной x_i заменяется на терм $f(x_1...x_{i-1})$, где f - новый m-арный функциональный символ; и т.д. для всех кванторов $\exists$.

Теорема. Если Φ - формула в п.н.ф., а Φ' - соответствующая ей формула в с.н.ф., то формулы Φ и Φ' одновременно невыполнимы.
Теорема. Замкнутая формула общезначима тогда и только тогда, когда ее отрицание невыполнимо.

Определение. Система дизъюнктов $S = \{D_1,...,D_n\}$ называется противоречивой, если $\forall$ I, $\exists$ $(a_1,...,a_n) \in D_I^n$ и $\exists$ $D_j \in S$, что D_j невыполнима на I.

Процесс доказательства общезначимости формулы Φ:

1) Построение п.н.ф. для $\neg \Phi$
2) Построение с.н.ф.
3) Построение набора дизъюнктов $S = \{D_1,...,D_n\}$ по с.н.ф.

4) Резолютивный вывод для S. Если вывод завершается успешно, то это означает, что S – противоречив, и, следовательно, $\neg \phi$ - невыполнима, а значит ϕ - общезначима.

Резолютивный вывод для набора дизъюнктов S состоит в построении конечной последовательности дизъюнктов $\{D_1', ..., D_k'\}$, такой что каждый D_i' получается из $\{D_1,...,D_n, D_1',...D_{i-1}'\}$ по правилам резолютивного вывода. Вывод считается успешным, если вывод заканчивается получением $D_k' = \square$ – пустого дизъюнкта.

Теорема (корректность резолютивного вывода). Если из S резолютивно выводим D, то S |= D.
Теорема (полнота резолютивного вывода). Если S – противоречив, то $\exists$ успешный резолютивный вывод для S.

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

Синтаксис логической программы

<программа>::=<программные утверждения><запрос>
<программное утверждение>::=<процедура>|<факт>
< процедура>::=<заголовок процедуры> :- <тело процедуры>
<заголовок процедуры> :- <атом>
<тело процедуры> :- <атом>|<атом>, <тело процедуры>
<факт> :- <атом>
<запрос>::= ?<тело процедуры>

Основная структура данных – список:

1) nil – константа, пустой список
2) *x*.nil – список, состоящий из единственного элемента *x*
3) *x*.*y* – список, состоящий из головного элемента *x* и хвоста *y*

Декларативная семантика

1) Процедура A_0 :- $A_1,...,A_n$ интерпретируется как $\forall x_1, ..., \forall x_n (A_1\&...\&A_n \rightarrow A_0)$, или, что то же самое, $\forall x_1, ..., \forall x_n (\neg A_1 \vee ... \vee \neg A_n \vee A_0)$
2) Факт B_0 :- интерпретируется как $\forall x_1, ..., \forall x_n B_0$
3) Запрос $?C_1,...,C_n$ интерпретируется как $\exists x_1,..., \exists x_n (C_1,...,C_n)$, где кванторы существования связывают все переменные

Ответ θ называется правильным, если P |= Gθ .

Определение. Ответом на запрос G к программе P называется подстановка $\theta = \{x_1/t_1,...,x_n/t_n\}$, где $x_1,...,x_n$ – переменные, входящие в запрос, а $t_1,...,t_n$ – основные термы.

Определение. Множество $H = H_0 \cup H_1 \cup ... \cup H_n \cup ...$ называется эрбрановским универсумом множества формул Г, если:

1) H_0 – множество всех констант из Г
2) $H_{i+1} = H \cup \{f(t_1,...,t_n | f$ – функции из Г, $t_1,...,t_n \in H_i\}$

Терм $P_i(t_1,...,t_n)$, где P_i – предикат из Г, а $t_1,...,t_n \in H$, называется основным термом. Эрбрановским базисом называется множество всех основных термов.
Определение. Эрбрановской интерпретацией для множества формул Г называется интерпретация I с областью H(Г), если:

1) для любой константы c из Г, $c_I = c$
2) для любой функции f из Г и для любых $t_1,...,t_n \in H$, $f(t_1,...,t_n)_I = f(t_1,..,t_n)$

Определение. Эрбрановская интерпретация I называется моделью для программы P, если I |= P.

Теорема (об основном правильном ответе). Пусть P – программа, A – атом с переменными $\{y_1,...,y_n\}$, $\theta = \{y_1/t_1,...,y_n/t_n$, где $t_1,...,t_n$ – основные термы. θ является правильным ответом на запрос ?A тогда и только тогда, когда A$\theta = M_P$, где M_P – наименьшая эрбрановская интерпретация, являющаяся моделью для P.

Операционная семантика

Определение. Подстановка θ называется унификатором выражений E_1 и E_2, если $E_1\theta = E_2\theta$. Унификатор η выражений E_1 и E_2 называется наиболее общим унификатором (НОУ), если для любого унификатора θ $\exists$ подстановка ρ , что $\theta = \eta\rho$, т.е. θ является суперпозицией подстановок η и ρ .
Определение. Пусть $G = ?C_1,...,C_{i-1}, C_i, C_{i+1},...,C_n$ – запрос к программе P, $D = A_0$:- $A_1,...,A_n$ – программное утверждение из P, причем множества переменных G и D не пересекаются, θ - НОУ для C_i и A_0. Тогда $G' = ?(C_1,...,C_{i-1},A_1,...,A_n,C_{i+1},...,C_n)\theta$ называется SLD-резольвентой запроса G к программе P с выделенной подцелью C_i и активизированными программным утверждением D и наиболее общим унификатором θ .

Определение. Пусть G_0 – запрос к программе P. Тогда последовательность (G_0, θ_0), (G_1, θ_1),...., (G_n, θ_n), где G_{i+1} – SLD-резольвента G_i, называется частичным SLD-вычислением ответа на запрос G_0 к программе P.

Определение. Частичное SLD-вычисление ответа на запрос G к программе P называется SLD-вычислением, если:

1) Частичное SLD-вычисление имеет вид (G_0, θ_0),...,(G_n, θ_n), □ и результатом выполнения запроса (G_n, θ_n) является пустой запрос. Это успешное SLD-вычисление
2) Частичное SLD-вычисление имеет вид (G_0, θ_0),...,(G_n, θ_n), ■ и ни одно выделение подцели G_n и ни одна активизация программного утверждение неприменимы для получения SLD-резольвенты. Такое вычисление называется тупиковым.
3) (G_0, θ_0),...,(G_n, θ_n),... – бесконечное вычисление

Определение. Пусть (G_0, θ_0),...,(G_n, θ_n), □ – успешное SLD-вычисление ответа на запрос G_0 к программе P, $\{y_1,...,y_n\}$ – множество переменных G_0. Тогда подстановка $\eta = \theta_0 ... \theta_n \mid y_1,...y_n$ – проекция композиции подстановок $\theta_0 ... \theta_n$ на переменные $y_1,...,y_n$ называется вычисленным ответом на G_0 к P.

Теорема (о корректности вычисленных ответов). Пусть θ - вычисленный ответ на запрос G к P. Тогда θ - правильный ответ.

Теорема (о полноте относительно вычисленных ответов). Пусть θ - правильный ответ на запрос G к P. Тогда $\exists$ вычисленный ответ λ, такой что $\exists$ подстановка ρ, что $\theta = \lambda \rho$.

Определение. Правилом вычисления R называется правило, по которому выбирается подцель C_i на каждом шаге SLD-вычисления запроса G к программе P.

Стандартное правило вычислений – выбирать всегда самую левую подцель.

Определение. Пусть G – запрос к программе P, R – правило вычислений. Деревом вычислений запроса G к программе P по правилу R называется корневое дерево, построенное по следующим правилам:

1) Корень дерева помечен исходным запросом.
2) Если вершина V помечена запросом G', а G_1",...,G_n" – всевозможные SLD-резольвенты, полученные из G' по правилу R с НОУ θ_1,..., θ_n соответственно, то из V выходит n дуг к вершинам U_1,...,U_n, помеченным G_1",...,G_n", а дуги помечены θ_1,..., θ_n.

Определение. Стратегией вычисления запроса G к программе P называется порядок обхода дерева SLD-вычислений.

Возможные стратегии обхода - в ширину и в глубину.

5. Транзакционное управление в СУБД. Методы сериализации транзакций.

Определение. Транзакция - это последовательность операций над БД, рассматриваемых СУБД как единое целое. Либо транзакция успешно выполняется, и СУБД фиксирует изменения БД, произведенные этой транзакцией, во внешней памяти, либо ни одно из этих изменений никак не отражается на состоянии БД. Понятие транзакции необходимо для поддержания логической целостности БД.

К транзакциям обычно предъявляются следующие требования:

1) atomicy – атомарность транзакции. Транзакция понимается как единица пользовательской активности по отношению к базе данных.
2) consistency – база данных находится в согласованном состоянии до начала транзакции и должна остаться в согласованном состоянии после успешного завершения транзакции, либо после ее отката.
3) isolation – отсутствие влияния различных транзакций друг на друга. Пользователь, работающий с СУБД, не должен ощущать на себе влияние работы других пользователей с той же базой данных.
4) durability – устойчивость внесенных изменений. Если транзакция была успешно выполнена и пользователь получил соответствующее подтверждение, то никакие дальнейшие события не могут отменить или исказить результаты этой транзакции.

Определение. Под сериализацией параллельно выполняющихся транзакций понимается такой порядок их работы, что суммарный эффект их смешанного выполнения эквивалентен эффекту некоторого последовательного их выполнения.

Метод сериализации, основанный на синхронизационных захватах

Наиболее распространенным в централизованных СУБД (включающих системы, основанные на архитектуре "клиент-сервер") является подход, основанный на соблюдении двухфазного протокола синхронизационных захватов объектов БД. В общих чертах протокол состоит в том, что перед выполнением любой операции в транзакции T над объектом базы данных r от имени транзакции T запрашивается синхронизационный захват объекта r в соответствующем режиме (в зависимости от вида операции).

Основными режимами синхронизационных захватов являются:

1) совместный режим - S (Shared), означающий разделяемый захват объекта и требуемый для выполнения операции чтения объекта;
2) монопольный режим - X (eXclusive), означающий монопольный захват объекта и требуемый для выполнения операций занесения, удаления и модификации.

Захваты объектов несколькими транзакциями по чтению совместимы, т.е. нескольким транзакциям допускается читать один и тот же объект, захват объекта одной транзакцией по чтению не совместим с захватом другой транзакцией того же объекта по записи, и захваты одного объекта разными транзакциями по записи не совместимы.

Для обеспечения сериализации транзакций синхронизационные захваты объектов, произведенные по инициативе транзакции, можно снимать только при ее завершении. Это требование порождает двухфазный протокол синхронизационных захватов. В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

1) первая фаза транзакции - накопление захватов;
2) вторая фаза (фиксация или откат) - освобождение захватов.

Объектами захвата могут служить:

1) файл - физический (с точки зрения базы данных) объект, область хранения нескольких отношений и, возможно, индексов;
2) отношение - логический объект, соответствующий множеству кортежей данного отношения;
3) страница данных - физический объект, хранящий кортежи одного или нескольких отношений, индексную или служебную информацию;
4) кортеж - элементарный физический объект базы данных.

Одним из наиболее чувствительных недостатков метода сериализации транзакций на основе синхронизационных захватов является возможность возникновение тупиков (deadlocks) между транзакциями. Поскольку тупики возможны, и никакого естественного выхода из тупиковой ситуации не существует, то эти ситуации необходимо обнаруживать и искусственно устранять.

Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) графа ожидания транзакций. Граф ожидания транзакций - это ориентированный двудольный граф, в котором существует два типа вершин - вершины, соответствующие транзакциям, и вершины, соответствующие объектам захвата. В этом графе существует дуга, ведущая из вершины-транзакции к вершине-объекту, если для этой транзакции существует удовлетворенный захват объекта. В графе существует дуга из вершины-объекта к вершине-транзакции, если транзакция ожидает удовлетворения захвата объекта. Легко показать, что в системе существует ситуация тупика, если в графе ожидания транзакций имеется хотя бы один цикл. Для разрешения тупика откатывается одна или несколько транзакций, так, чтобы остальные могли продолжить свою работу.

Метод сериализации, основанный на временных метках

Альтернативный метод сериализации транзакций, хорошо работающий в условиях редких конфликтов транзакций и не требующий построения графа ожидания транзакций, основан на использовании временных меток.

Основная идея метода (у которого существует множество разновидностей) состоит в следующем: если транзакция T_1 началась раньше транзакции T_2, то система обеспечивает такой режим выполнения, как если бы T1 была целиком выполнена до начала T_2. Для этого каждой транзакции T предписывается временная метка t, соответствующая времени начала T. При выполнении операции над объектом r транзакция T помечает его своей временной меткой и типом операции (чтение или изменение).

Перед выполнением операции над объектом r транзакция T_1 выполняет следующие действия:

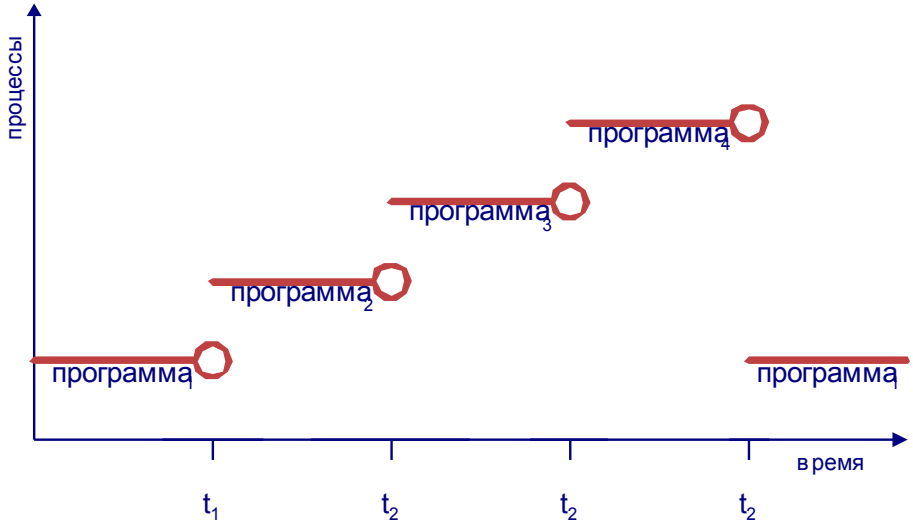
1) Проверяет, не закончилась ли транзакция T, пометившая этот объект. Если T закончилась, T_1 помечает объект r и выполняет свою операцию.
2) Если транзакция T не завершилась, то T_1 проверяет конфликтность операций. Если операции неконфликтны, при объекте r остается или проставляется временная метка с меньшим значением, и транзакция T1 выполняет свою операцию.
3) Если операции T_1 и T конфликтуют, то если $t(T) > t(T_1)$ (т.е. транзакция T является более "молодой", чем T_1), производится откат T и T_1 продолжает работу.
4) Если же $t(T) < t(T_1)$ (T "старше" T_1), то T_1 получает новую временную метку и начинается заново.

К недостаткам метода временных меток относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов. Это связано с тем, что конфликтность транзакций определяется более грубо. Кроме того, в распределенных системах не очень просто вырабатывать глобальные временные метки с отношением полного порядка.

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

Если система обрабатывает единственную программу, то в это время ЦП не производит никакой полезной работы, то есть простаивает (на самом деле термин простой достаточно условный, так как при этом работает операционная система).

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



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

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

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

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

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

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

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

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

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

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

Взаимодействие процессов. Методы синхронизации

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

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

Необходимо уметь решать две важнейшие задачи:

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

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

Рассмотрим ситуацию, изображенную на Рис. 1:

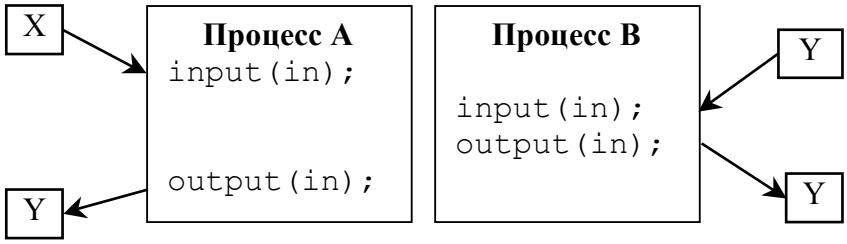


Рис. 1 Конкуренция процессов за ресурс.

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

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

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

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

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

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

Рассмотри классические методы (средства) синхронизации.

Семафоры Дейкстры

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

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

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

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

Мониторы Хоара

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

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

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

Классические задачи синхронизации процессов
«Обедающие философы»

"Обедающие философы"

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

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

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

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

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

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

```
   # define EATING 2                    /* философ ест */

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

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

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

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

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

Задача «читателей и писателей»

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

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

void reader (void)
{
      while (TRUE)                  /* бесконечный цикл */
      {
            down (&mutex);    /* получить эксклюзивный доступ к «rc»*/
```

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

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

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

Надо заметить, что приведенный алгоритм дает преимущество при доступе к базе данных процессам-читателям, т.к. процесс, ожидающий доступа по записи, будет ждать до тех пор, пока все читающие процессы не окончат работу, и если в это время появляется новый читающий процесс, он тоже беспрепятственно получит доступ. Это может привести к неприятной ситуации в том случае, если в фазе, когда ресурс доступен по чтению, и имеется ожидающий процесс-писатель, будут появляться новые и новые читающие процессы. Чтобы этого избежать, можно модифицировать алгоритм таким образом, чтобы в случае, если имеется хотя бы один ожидающий процесс-писатель, новые процессы-читатели не получали доступа к ресурсу, а ожидали, когда процесс-писатель обновит данные. В этой ситуации процесс-писатель должен будет ожидать окончания работы с ресурсом только тех читателей, которые получили доступ раньше, чем он его запросил. Однако, обратная сторона данного решения в том, что оно несколько снижает производительность процессов-читателей, т.к. вынуждает их ждать в тот момент, когда ресурс не занят в эксклюзивном режиме.

Задача о «спящем парикмахере»

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

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

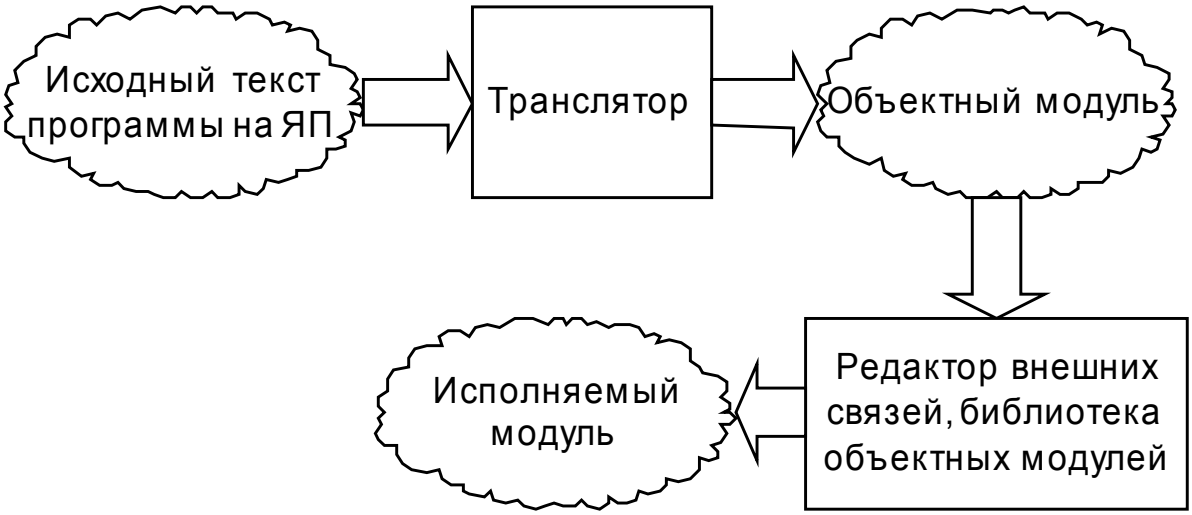
```
#define CHAIRS 5
typedef int semaphore;        /* некий семафор */
```

```
semaphore customers = 0;      /* посетители, ожидающие в очереди */
semaphore barbers = 0;  /* парикмахеры, ожидающие посетителей */
                                    semaphore mutex = 1;    /* контроль за доступом к
                                    переменной waiting */
int waiting = 0;

void barber()
{
while (true) {
                          down(customers);  /* если customers == 0, т.е. посетителей нет, то
                          заблокируемся до появления посетителя */
      down(&mutex);      /* получаем доступ к waiting */
      waiting = wating - 1;   /* уменьшаем кол-во ожидающих клиентов */
      up(&barbers);           /* парикмахер готов к работе */
      up(&mutex);             /* освобождаем ресурс waiting */
      cut_hair();             /* процесс стрижки */
};}

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

8. Виртуальная память. Модели организации оперативной памяти.
Аппарат виртуальной памяти

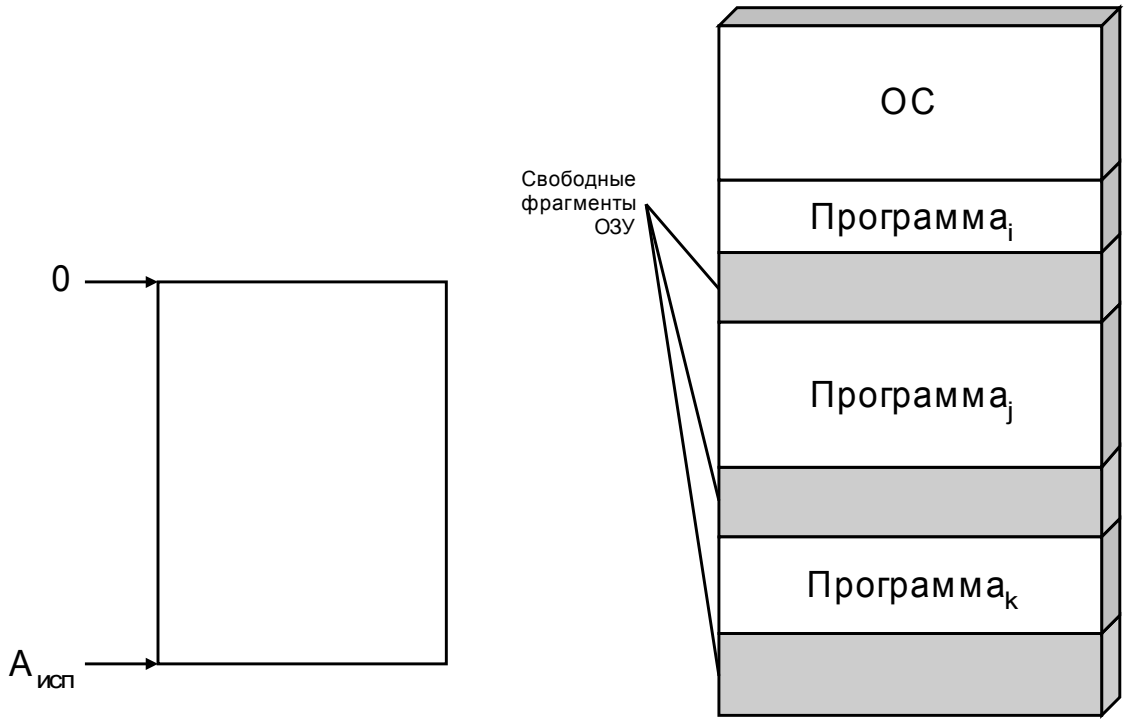
Рассмотрим некоторые проблемы организации адресации в программах/процессах и связанные с ними проблемы использования ОЗУ в целом.

В общем случае схема получения исполняемого кода программы следующая:



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

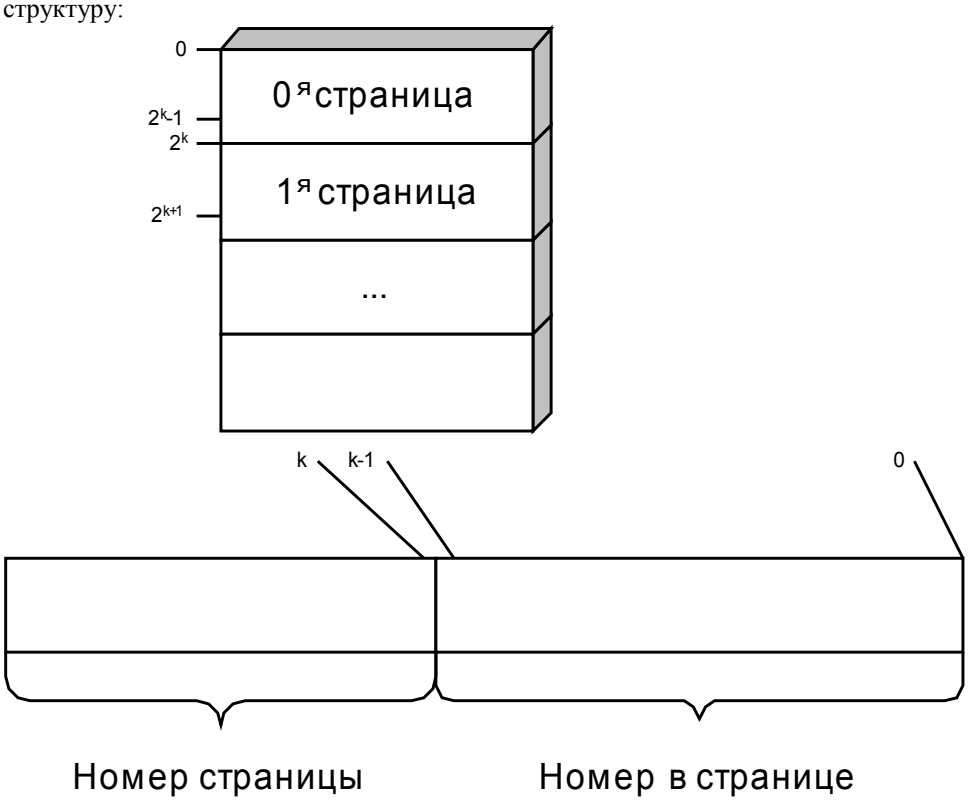
Элементарное программно-аппаратное решение – использование возможности ***базирования адресов***. Суть его состоит в следующем: пусть имеется исполняемый программный модуль. Виртуальное адресное пространство этого модуля лежит в диапазоне $[0, A_{кон}]$. В ЭВМ выделяется специальный регистр базирования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



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

Организация страничной памяти

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



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

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

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

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

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

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

Таблица содержит m строк. Содержимое таблицы определяет соответствие виртуальной памяти физической для выполняющейся в данный момент программы/процесса. Соответствие определяется следующим образом: i-я строка таблицы соответствует i-й виртуальной странице.
Содержимое строки α_i определяет, чему соответствует i-я виртуальная страница программы/процесса. Если $\alpha_i \geq 0$, то это означает, что α_i есть номер физической страницы, которая соответствует виртуальной странице программы/процесса. Если $\alpha_i = -1$, то это означает, что для i-й виртуальной страницы нет соответствия физической странице ОЗУ (обработка этой ситуации ниже).

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

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

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

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

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

Недостаток – необходимость наличия в ЦП аппаратной таблицы значительных размеров.

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

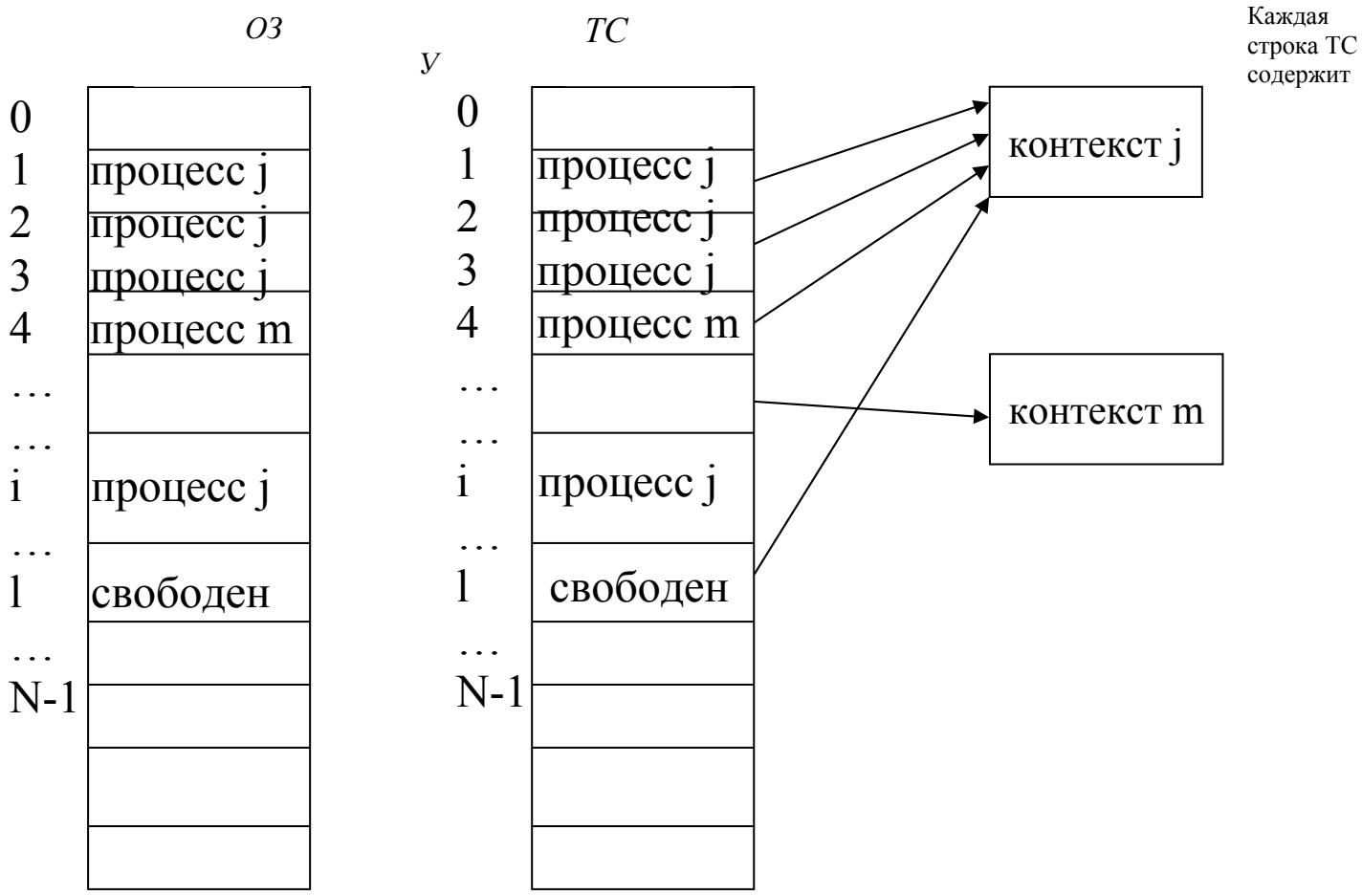
Управление оперативной памятью

Решение следующих задач:

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

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

Операционная система формирует таблицу страниц (ТС):



информацию о статусе соответствующей физической страницы:

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

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

При поступлении процесса в БОП заполняется ТСП. В начальный момент из описателей процесса, сформированных на этапе обработки в БВП выбирается список виртуальных страниц, который размещается в ТСП. Затем ОС анализирует содержимое

ТС и «приписывает» виртуальным страницам их физические эквиваленты (при этом идет загрузка содержимого соответствующих виртуальных страниц из внешней памяти в физические страницы ОЗУ).

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

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

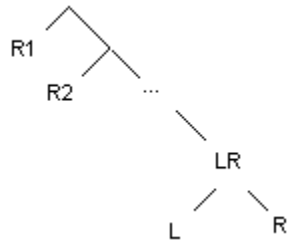
В рассмотренном примере затронуты элементы решения задач распределения физической памяти, поддержки использования аппарата виртуальной памяти, подкачки страниц. На самом деле в реальности полная логика действий существенно сложнее и окончательные настройки аппаратных средств (виртуальной памяти, защиты)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

СВОППИНГ

ЗАЩИТА ПАМЯТИ

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

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.



Предположим сначала, что распределение регистров осуществляется по простейшей схеме слева-направо. 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 $n+n_l+1$. 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 n_r регистров будет использовано для вычисления R. Всего будет использовано $n+n_r+1$ регистров. 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.

Алгоритм работает следующим образом. Сначала осуществляется разметка синтаксического дерева по следующим правилам:

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 большее из чисел l_1 или l_2 либо число l_1+1, если $l_1=l_2$.

Эта разметка позволяет определить, какое из поддеревьев требует большего количества регистров для своего вычисления.

Затем осуществляется распределение регистров для результатов операций:

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).
3) Если же метка левого потомка больше или равна метке правого потомка, то наоборот, сначала вычисляется левое поддерево и его результат помещается в регистр R. После этого формируется код по следующим правилам.

Правила генерации кода:

1) если вершина - правый лист с меткой 1, то ей соответствует код "LOAD 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 "Op X,R", где снова 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 метки левой, то вершине соответствует код "Op R+1,R", где R – регистр назначенный левой вершине, а R+1 – правой.
4) Если метка правой вершины меньше или равна метке левой вершины, то вершине соответствует код "Op R,R+1; MOVE R+1,R", где R – регистр назначенный левой вершине, а R+1 – правой. Последняя команда генерируется для того, чтобы получить результат в нужном регистре (в случае коммутативной операции операнды операции можно поменять местами и избежать дополнительной пересылки).

10. Основные принципы объектно-ориентированного программирования.

Определение. Объектно-ориентированное программирование - это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследования.

Ключевыми отличиями ООП от других методологий программирования являются следующие отличия:

1) ООП использует в качестве элементов конструкции объекты, а не алгоритмы
2) Каждый объект является реализацией какого-либо определенного типа (класса)
3) Классы организованы иерархически

При несоблюдении хотя бы одного из указанных требований программа перестает быть ОО. (В частности при нарушении 3 имеем программирование на основе Абстрактных Типов Данных)

Определение. Язык программирования называется объектно-ориентированным тогда и только тогда, когда выполнены следующие условия:

1) Имеется поддержка объектов в виде абстракции данных имеющих интерфейсную часть в виде поименованных операций и защищенную область локальных данных
2) Объекты относятся к соответствующим типам (классам)
3) Классы могут наследовать атрибуты и методы от суперклассов (базовых классов)
4) Имеется поддержка полиморфных функций

Объектно-ориентированный подход основывается на следующих элементах:

1) Абстрагирование
2) Инкапсуляция (ограничение доступа)
3) Иерархия (в частности наследование)
4) Полиморфизм

Абстрагирование

Определение. Абстракция - это такие существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов, и, таким образом четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от деталей их осуществления.

ОО стиль программирования связан с воздействием на объекты (например передача ему сообщения). Путем воздействия на объект вызывается определенная реакция этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия составляют характер поведения объекта.

Инкапсуляция (ограничение доступа)

Определение. Ограничение доступа - это процесс защиты отдельных элементов объекта, не затрагивающий существенных характеристик объекта как целого.

Ограничение доступа позволяет вносить в программу изменения, сохраняя ее надежность и позволяя минимизировать затраты на этот процесс.

Абстрагирование и ограничение доступа дополняют друг друга: абстрагирование фокусирует внимание на внешних особенностях объекта, а ограничение доступа не позволяет объектам-пользователям различать внутреннее устройство объекта.

В описании класса можно выделить две части:

1) Интерфейс, который отражает внешнее проявление объекта, создавая абстракцию поведения всех объектов данного класса. В интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами.
2) Реализация, которая описывает механизмы достижения желаемого поведения объекта. Реализация скрывает от других объектов все детали, не имеющие отношение к процессу взаимодействия объектов.

Изменение реализации, вообще говоря, не влечет за собой изменение интерфейса.

Иерархия

Определение. Иерархия в ОО - это ранжированная или упорядоченная иерархия абстракций.

Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия по номенклатуре) и структура объектов (иерархия по составу).

Примеры иерархий:

1) Наследование означает такое соотношение между классами, когда один класс использует структурную или функциональную часть одного или нескольких других классов (соответственно простое или множественное наследование). Иными словами, наследование - такая иерархия абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов. (Иерархия обобщение - специализация)
2) Агрегирование (отношение по составу). Объект состоит из подобъектов.

Принципы абстрагирования, ограничения доступа и иерархии конкурируют между собой. Абстрагирование данных состоит в установлении жестких границ, защищающих состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных классов. В связи с этим интерфейсная часть класса может быть разделена на три части:

1) Обособленную (private) - видимая только для самого класса
2) Защищенную (protected) - видимую также и для подклассов
3) Общедоступную (public) - видимую для всех

Полиморфизм

Определение. Полиморфизм – это способность операции (функции) с одним и тем же именем выполнять различные действия в зависимости от типа своих операндов.

Полиморфизм может быть статическим и динамическим:

1) Статический - перекрытие операций (Ада, C++,Object Pascal).
2) Динамический - механизм виртуальных функций

Виртуальные функции являются примером полиморфных функций. Виртуальная функция может быть переопределена в производном классе, следовательно ее реализация зависит от всей последовательности методических описаний и наследственной иерархии. Какая именно из виртуальных функций будет вызвана - зависит от динамического типа объекта и определяется в момент обращения к виртуальной функции. Для этого используется таблица виртуальных функций, определенная для каждого класса.

Дополнительные возможности ОО языков:

Некоторые языки позволяют определять несколько специальных методов класса:

1) Конструктор - специальная процедура класса для создания и/или инициализации начального состояния объекта. В частности, конструктор может инициализировать таблицу виртуальных функций.
2) Деструктор - специальная процедура класса, которая делает состояние объекта неопределенным и (или) ликвидирует сам объект.

Некоторые преимущества ОО подхода:

1) Использование объектного подхода существенно повышает качество разработки в целом и ее фрагментов. ОО Системы часто получаются более компактными чем их не-ОО аналоги
2) Использование объектного подхода приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к ее полной переработке в случае существенных изменений исходных требований
3) Ориентирован на человеческое восприятие мира

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

Структура компилятора

Обобщенная структура компилятора и основные фазы компиляции показаны на рис. 1.1.

Рис. 1.1:

На 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 (идентификаторов, строк, чисел и т.д.), 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 др.).

Основная задача синтаксического анализа - разбор структуры программы. Как правило, под структурой понимается дерево, соответствующее разбору в контекстно-свободной 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. В процессе синтаксического анализа также обнаруживаются ошибки, связанные со структурой программы.
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, использующимся 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.
Фаз оптимизации может быть несколько. Оптимизации обычно делят на машинно-зависимые и машинно-независимые, локальные и глобальные. 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 т.д.
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.

13. Построение канонического множества LR(1) ситуаций и таблиц действии и переходов для LR(1) грамматик.

Определение. Пусть G=<N,T,P,S> - контекстно-свободная грамматика, где N - множество нетерминальных символов, T - множество терминальных символов, P - множество правил вывода и S - аксиома. Будем говорить, что uxv выводится за один шаг из uAv (и записывать это как uAv$\Rightarrow$ uxv), если A$\to$ x - правило вывода и u и v - произвольные строки из (N$\cup$ T)*. Если $u_1\Rightarrow u_2\Rightarrow \ldots\Rightarrow u_n$, будем говорить, что из u_1 выводится u_n, и записывать это как $u_1\Rightarrow {}^*u_n$. Т.е.:

1) u$\Rightarrow$ *u для любой строки u,
2) если u$\Rightarrow$ *v и v$\Rightarrow$ *w, то u$\Rightarrow$ *w.

Аналогично, "$\Rightarrow$ $^+$" означает "выводится за один или более шагов".

Определение. Языком L(G), порожденным грамматикой G с начальным символом S называется множество L(G) = {w| w содержит только терминальные символы и S$\Rightarrow$ $^+$w}. Строка w называется предложением в G.
Определение. Если S$\Rightarrow$ *u, где u может содержать нетерминалы, то u называется сентенциальной формой в G. Предложение - это сентенциальная форма, не содержащая нетерминалов.
Определение. Рассмотрим выводы, в которых в любой сентенциальной форме на каждом шаге делается подстановка самого левого нетерминала. Такой вывод называется левосторонним. Если S$\Rightarrow$ *u в процессе левостороннего вывода, то u - левая сентенциальная форма. Аналогично определяется правосторонний вывод.
Определение. Упорядоченным графом называется пара (V,E), где V обозначает множество вершин, а E - множество линейно упорядоченных списков дуг, каждый элемент которого имеет вид $((v,e_1),(v,e_2),\ldots,(v,e_n))$. Этот элемент указывает, что из вершины a выходят n дуг, причем первой из них считается дуга, входящая в вершину e_1, второй - дуга, входящая в вершину e_2, и т.д.
Определение. Упорядоченное помеченное дерево D называется деревом вывода (или деревом разбора) в КС-грамматике G(S)=(N,T,P,S), если выполнены следующие условия:

1) корень дерева D помечен S;
2) каждый лист помечен либо a$\in$ T, либо e;
3) каждая внутренняя вершина помечена нетерминалом;
4) если N - нетерминал, которым помечена внутренняя вершина и $X_1,\ldots,X_n$ - метки ее прямых потомков в указанном порядке, то N$\to$ $X_1\ldots X_k$ - правило из множества P.

Определение. Автомат с магазинной памятью (сокращенно МП-автомат) - это семерка $P=(Q,T,\Gamma,D,q_0,Z_0,F)$, где

1) Q - конечное множество символов состояний, представляющих всевозможные состояния управляющего устройства;
2) T - конечный входной алфавит.
3) Γ - конечный алфавит магазинных символов.

4) D - функция переходов - отображение множества Qx(T ∪ {e})xГ в множество конечных подмножеств QxГ*.
5) $q_0 \in Q$ - начальное состояние управляющего устройства.
6) $Z_0 \in Г$ - символ, находящийся в магазине в начальный момент (начальный символ).
7) $F \subseteq Q$ - множество заключительных состояний.

Определение. Конфигурацией МП-автомата называется тройка $(q,w,u) \in QxT^*xГ^*$, где

1) q - текущее состояние управляющего устройства;
2) w - неиспользованная часть входной цепочки; первый символ цепочки w находится под входной головкой; если w = e, то считается, что вся входная лента прочитана;
3) u - содержимое магазина; самый левый символ цепочки u считается верхним символом магазина; если u = e, то магазин считается пустым.

Определение. Такт работы МП-автомата P будем представлять в виде бинарного отношения $\hbar$, определенного на конфигурациях. Будем писать $(q,aw,Zu)\ \hbar\ (q',w,vu)$
если множество D(q,a,Z) содержит (q',v), где $q, q' \in Q$, $a \in T \cup \{e\}$, $w \in T^*$, $Z \in Г$, $u,v \in Г^*$.

Определение. Начальной конфигурацией МП-автомата P называется конфигурация вида (q_0,w,Z_0), где $w \in T^*$, т.е. управляющее устройство находится в начальном состоянии, входная лента содержит цепочку, которую нужно распознать, а в магазине есть только начальный символ Z_0. Заключительная конфигурация - это конфигурация вида (q,e,u), где $q \in F$, $u \in Г^*$.

Определение. Говорят, что цепочка w допускается МП-автоматом P, если $(q_0,w,Z_0)\ \hbar\ ^*(q,e,u)$ для некоторых $q \in F$ и $u \in Г^*$. Языком, определяемым (или допускаемым) автоматом P (обозначается L(P)), называют множество цепочек, допускаемых автоматом P.

Разбор снизу-вверх типа сдвиг-свертка

В процессе разбора снизу-вверх типа сдвиг-свертка строится дерево разбора входной строки, начиная с листьев (снизу) к корню (вверх). Этот процесс можно рассматривать как "свертку" строки w к начальному символу грамматики. На каждом шаге свертки подстрока, которую можно сопоставить правой части некоторого правила вывода, заменяется символом левой части этого правила вывода, и если на каждом шаге выбирается правильная подстрока, то в обратном порядке прослеживается правосторонний вывод.

Определение. Подстрока сентенциальной формы, которая может быть сопоставлена правой части некоторого правила вывода, свертка по которому к левой части правила соответствует одному шагу в обращении правостороннего вывода, называется *основой* строки.

Самая левая подстрока, которая сопоставляется правой части некоторого правила вывода $A \to v$, не обязательно является основой, поскольку свертка по правилу $A \to v$ может дать строку, которая не может быть сведена к аксиоме.

Определение. Замена основы в сентенциальной форме на нетерминал левой части называется отсечением основы.

Таким образом, главная задача анализатора типа сдвиг-свертка - это выделение и отсечение основы.

LR(k)-анализаторы

LR-анализатор состоит из входа, выхода, магазина, управляющей программы и таблицы анализа, которая имеет две части - действий и переходов. Управляющая программа одна и та же для всех анализаторов, разные анализаторы различаются таблицами анализа. Программа анализатора читает символы из входного буфера 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, называемый состоянием.

Таблица анализа состоит из двух частей: действия (action) и переходов (goto).

Определение. Конфигурацией LR анализатора называется пара, первая компонента которой - содержимое магазина, а вторая - непросмотренный вход:

$(S_0X_1S_1X_2S_2 \dots X_mS_m, a_i\ a_{i+1} \dots a_n\ \$)$

Эта конфигурация соответствует правой сентенциальной форме

$X_1X_2 \dots X_ma_ia_{i+1}...a_n$

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.

Очередной шаг анализатора определяется текущим входным символом a_i и символом состояния на верхушке магазина S_m. Элемент таблицы действий action$[S_m,a_i]$ для состояния S_m и входа a_i, может иметь одно их четырех значений:

1) shift S, сдвиг, где S - состояние,
2) reduce $A \to w$, свертка по правилу грамматики $A \to w$,
3) accept, допуск,
4) error, ошибка.

Схема работы автомата на каждом шаге следующая:

1) Если action$[S_m,a_i]$=shift S, то анализатор выполняет шаг сдвига, переходя в конфигурацию $(S_0X_1S_1 \dots X_mS_ma_iS, a_{i+1} \dots a_n\$)$
2) Если action$[S_m,a_i]$=reduce $A \to w$, то анализатор выполняет свертку, переходя в конфигурацию $(S_0X_1S_1 \dots X_{m-r}S_{m-r}AS, a_ia_{i+1}...a_n\$)$, где S=goto$[S_{m-r},A]$ и r - длина w, правой части правила вывода.
3) Если action$[S_m,a_i]$=accept, то автомат останавливается с успешным результатом
4) Если action$[S_m,a_i]$=error, то автомат останавливается с ошибкой.

14. Архитектура параллельных вычислительных систем.

Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и собственно параллельность. Оба вида параллельной обработки интуитивно понятны, поэтому сделаем лишь небольшие пояснения.

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.

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).

Казалось бы, конвейерную обработку можно с успехом заменить обычным параллелизмом, для чего продублировать основное устройство столько раз, сколько ступеней конвейера предполагается выделить. В самом деле, пять устройств предыдущего примера обработают 100 пар аргументов за 100 единиц времени, что быстрее времени работы конвейерного устройства! В чем же дело? Ответ прост, увеличив в пять раз число устройств, мы значительно увеличиваем как объем аппаратуры, так и ее стоимость.

Классификация Флинна

Самой ранней и наиболее известной является классификация архитектур вычислительных систем, предложенная в 1966 году М.Флинном. Классификация базируется на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

1) SISD (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных. К этому классу относятся, прежде всего, классические последовательные машины, или иначе, машины фон-неймановского типа, например, PDP-11 или VAX 11/780. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом и каждая команда инициирует одну операцию с одним потоком данных. Не имеет значения тот факт, что для увеличения скорости обработки команд и скорости выполнения арифметических операций может применяться конвейерная обработка - как машина CDC 6600 со скалярными функциональными устройствами, так и CDC 7600 с конвейерными попадают в этот класс.
2) SIMD (single instruction stream / multiple data stream) - одиночный поток команд и множественный поток данных. В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными - элементами вектора. Способ выполнения векторных операций не оговаривается, поэтому обработка элементов вектора может производиться либо процессорной матрицей, как в ILLIAC IV, либо с помощью конвейера, как, например, в машине CRAY-1.
3) MISD (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не смогли представить убедительный пример реально существующей вычислительной системы, построенной на данном принципе.
4) MIMD (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.

Предложенная схема классификации вплоть до настоящего времени является самой применяемой при начальной характеристике того или иного компьютера. Если говорится, что компьютер принадлежит классу SIMD или MIMD, то сразу становится понятным базовый принцип его работы, и в некоторых случаях этого бывает достаточно. Однако видны и явные недостатки. В частности, некоторые заслуживающие внимания архитектуры, например dataflow и векторно-конвейерные машины, четко не вписываются в данную классификацию. Другой недостаток - это чрезмерная заполненность класса MIMD. Необходимо средство, более избирательно систематизирующее архитектуры, которые по Флинну попадают в один класс, но совершенно различны по числу процессоров, природе и топологии связи между ними, по способу организации памяти и, конечно же, по технологии программирования.

Массивно-параллельные системы (MPP)

Система состоит из однородных вычислительных узлов, включающих:

1) один или несколько центральных процессоров (обычно RISC),

2) локальную память (прямой доступ к памяти других узлов невозможен),
3) коммуникационный процессор или сетевой адаптер
4) иногда - жесткие диски (как в SP) и/или другие устройства В/В

К системе могут быть добавлены специальные узлы ввода-вывода и управляющие узлы. Узлы связаны через некоторую коммуникационную среду (высокоскоростная сеть, коммутатор и т.п.)
Общее число процессоров в реальных системах достигает нескольких тысяч (ASCI Red, Blue Mountain).
Существуют два основных варианта ОС для управления такими компьютерами:

1) Полноценная ОС работает только на управляющей машине (front-end), на каждом узле работает сильно урезанный вариант ОС, обеспечивающие только работу расположенной в нем ветви параллельного приложения. Пример: Cray T3E.
2) На каждом узле работает полноценная UNIX-подобная ОС (вариант, близкий к кластерному подходу). Пример: IBM RS/6000 SP + ОС AIX, устанавливаемая отдельно на каждом узле.

Программирование в рамках модели передачи сообщений (MPI, PVM, BSPlib)

Примеры: IBM RS/6000 SP2, Intel PARAGON/ASCI Red, SGI/CRAY T3E, Hitachi SR8000, транспьютерные системы Parsytec.

Симметричные мультипроцессорные системы (SMP)

Система состоит из нескольких однородных процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти либо с помощью общей шины (базовые 2-4 процессорные SMP-сервера), либо с помощью crossbar-коммутатора (HP 9000). Аппаратно поддерживается когерентность кэшей.
Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает сильные ограничения на их число - не более 32 в реальных системах. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры.
Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически (в процессе работы) распределяет процессы/нити по процессорам (scheduling), но иногда возможна и явная привязка.
Программирование в модели общей памяти. (POSIX threads, OpenMP). Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания.

Примеры: HP 9000 V-class, N-class; SMP-сервера и рабочие станции на базе процессоров Intel (IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu и др.).

Системы с неоднородным доступом к памяти (NUMA)

Система состоит из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора. Поддерживается единое адресное пространство, аппаратно поддерживается доступ к удаленной памяти, т.е. к памяти других модулей. При этом доступ к локальной памяти в несколько раз быстрее, чем к удаленной. В случае, если аппаратно поддерживается когерентность кэшей во всей системе (обычно это так), говорят об архитектуре cc-NUMA (cache-coherent NUMA)
Масштабируемость NUMA-систем ограничивается объемом адресного пространства, возможностями аппаратуры поддежки когерентности кэшей и возможностями операционной системы по управлению большим числом процессоров. На настоящий момент, максимальное число процессоров в NUMA-системах составляет 256 (Origin2000).
Обычно вся система работает под управлением единой ОС, как в SMP. Но возможны также варианты динамического "подразделения" системы, когда отдельные "разделы" системы работают под управлением разных ОС (например, Windows NT и UNIX в NUMA-Q 2000).
Программирование аналогично SMP.

Примеры: HP HP 9000 V-class в SCA-конфигурациях, SGI Origin2000, Sun HPC 10000, IBM/Sequent NUMA-Q 2000, SNI RM600.

Параллельные векторные системы (PVP)

Основным признаком PVP-систем является наличие специальных векторно-конвейерных процессоров, в которых предусмотрены команды однотипной обработки векторов независимых данных, эффективно выполняющиеся на конвейерных функциональных устройствах. Как правило, несколько таких процессоров (1-16) работают одновременно над общей памятью (аналогично SMP) в рамках многопроцессорных конфигураций. Несколько таких узлов могут быть объединены с помощью коммутатора (аналогично MPP).
Эффективное программирование подразумевает векторизацию циклов (для достижения разумной производительности одного процессора) и их распараллеливание (для одновременной загрузки нескольких процессоров одним приложением).

Примеры: NEC SX-4/SX-5, линия векторно-конвейерных компьютеров CRAY: от CRAY-1, CRAY J90/T90, CRAY SV1, серия Fujitsu VPP.

Кластерные системы

Набор рабочих станций (или даже ПК) общего назначения, используется в качестве дешевого варианта массивно-параллельного компьютера. Для связи узлов используется одна из стандартных сетевых технологий (Fast/Gigabit Ethernet, Myrinet) на базе шинной архитектуры или коммутатора. При объединении в кластер компьютеров разной мощности или разной архитектуры, говорят о гетерогенных (неоднородных) кластерах.
Узлы кластера могут одновременно использоваться в качестве пользовательских рабочих станций. В случае, когда это не нужно, узлы могут быть существенно облегчены и/или установлены в стойку.
Используются стандартные для рабочих станций ОС, чаще всего, свободно распространяемые - Linux/FreeBSD, вместе со специальными средствами поддержки параллельного программирования и распределения нагрузки.
Программирование, как правило, в рамках модели передачи сообщений (чаще всего - MPI). Дешевизна подобных систем оборачивается большими накладными расходами на взаимодействие параллельных процессов между собой, что сильно сужает потенциальный класс решаемых задач.

Примеры: NT-кластер в NCSA, Beowulf-кластеры.

Историческая справка

IBM 701 (1953), IBM 704 (1955): разрядно-параллельная память, разрядно-параллельная арифметика.
Все самые первые компьютеры (EDSAC, EDVAC, UNIVAC) имели разрядно-последовательную память, из которой слова считывались последовательно бит за битом. Первым коммерчески доступным компьютером, использующим разрядно-параллельную память (на CRT) и разрядно-параллельную арифметику, стал IBM 701, а наибольшую популярность получила модель IBM 704 (продано 150 экз.), в которой, помимо сказанного, была впервые применена память на ферритовых сердечниках и аппаратное АУ с плавающей точкой.

IBM 709 (1958): независимые процессоры ввода/вывода.
Процессоры первых компьютеров сами управляли вводом/выводом. Однако скорость работы самого быстрого внешнего устройства, а по тем временам это магнитная лента, была в 1000 раз меньше скорости процессора, поэтому во время операций ввода/вывода процессор фактически простаивал. В 1958г. к компьютеру IBM 704 присоединили 6 независимых процессоров ввода/вывода, которые после получения команд могли работать параллельно с основным процессором, а сам компьютер переименовали в IBM 709. Данная модель получилась удивительно удачной, так как вместе с модификациями было продано около 400 экземпляров, причем последний был выключен в 1975 году - 20 лет существования!

IBM STRETCH (1961): опережающий просмотр вперед, расслоение памяти.
В 1956 году IBM подписывает контракт с Лос-Аламосской научной лабораторией на разработку компьютера STRETCH, имеющего две принципиально важные особенности: опережающий просмотр вперед для выборки команд и расслоение памяти на два банка для согласования низкой скорости выборки из памяти и скорости выполнения операций.

ATLAS (1963): конвейер команд.
Впервые конвейерный принцип выполнения команд был использован в машине ATLAS, разработанной в Манчестерском университете. Выполнение команд разбито на 4 стадии: выборка команды, вычисление адреса операнда, выборка операнда и выполнение операции. Конвейеризация позволила уменьшить время выполнения команд с 6 мкс до 1,6 мкс. Данный компьютер оказал огромное влияние, как на архитектуру ЭВМ, так и на программное обеспечение: в нем впервые использована мультипрограммная ОС, основанная на использовании виртуальной памяти и системы прерываний.

CDC 6600 (1964): независимые функциональные устройства.
Фирма Control Data Corporation (CDC) при непосредственном участии одного из ее основателей, Сеймура Р.Крэя (Seymour R.Cray) выпускает компьютер CDC-6600 - первый компьютер, в котором использовалось несколько независимых функциональных устройств.

CDC 7600 (1969): конвейерные независимые функциональные устройства.
CDC выпускает компьютер CDC-7600 с восемью независимыми конвейерными функциональными устройствами - сочетание параллельной и конвейерной обработки.

ILLIAC IV (1974): матричные процессоры.
Проект: 256 процессорных элементов (ПЭ) = 4 квадранта по 64ПЭ, возможность реконфигурации: 2 квадранта по 128ПЭ или 1 квадрант из 256ПЭ, такт 40нс, производительность 1Гфлоп;

CRAY 1 (1976): векторно-конвейерные процессоры
В 1972 году С.Крэй покидает CDC и основывает свою компанию Cray Research, которая в 1976г. выпускает первый векторно-конвейерный компьютер CRAY-1: время такта 12.5нс, 12 конвейерных функциональных устройств, пиковая производительность 160 миллионов операций в секунду, оперативная память до 1Мслова (слово - 64 разряда), цикл памяти 50нс. Главным новшеством является введение векторных команд, работающих с целыми массивами независимых данных и позволяющих эффективно использовать конвейерные функциональные устройства.

Конвейерность и параллелизм

При параллелизме совмещение операций достигается путем воспроизведения в нескольких копиях аппаратной структуры. Высокая производительность достигается за счет одновременной работы всех элементов структур, осуществляющих решение различных частей задачи.

Конвейеризация (или конвейерная обработка) в общем случае основана на разделении подлежащей исполнению функции на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры.
Выполнение типичной команды можно разделить на следующие этапы:

1) выборка команды - IF (по адресу, заданному счетчиком команд, из памяти
2) извлекается команда);
3) декодирование команды / выборка операндов из регистров - ID;
4) выполнение операции / вычисление эффективного адреса памяти - EX;
5) обращение к памяти - MEM;
6) запоминание результата - WB.

Чтобы конвейеризовать эту схему, мы можем просто разбить выполнение команд на указанные выше этапы, отведя для выполнения каждого этапа один такт синхронизации, и начинать в каждом такте выполнение новой команды.

При реализации конвейерной обработки возникают ситуации, которые препятствуют выполнению очередной команды из потока команд в предназначенном для нее такте. Такие ситуации называются конфликтами. Конфликты снижают реальную производительность конвейера, которая могла бы быть достигнута в идеальном случае.

Существуют три класса конфликтов:

1) Структурные конфликты, которые возникают из-за конфликтов по ресурсам, когда аппаратные средства не могут поддерживать все возможные комбинации команд в режиме одновременного выполнения с совмещением.
2) Конфликты по данным, возникающие в случае, когда выполнение одной команды зависит от результата выполнения предыдущей команды.
3) Конфликты по управлению, которые возникают при конвейеризации команд переходов и других команд, которые изменяют значение счетчика команд.

15. Технологии параллельного программирования.

Краткая характеристика MPI.

MPI представляет собой программный инструмент, предназначенных для поддержки работы параллельных процессов в терминах передачи сообщений для обеспечения связи между ветвями параллельного приложения. MPI предоставляет программисту единый механизм взаимодействия ветвей внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные / многопроцессорные с общей/раздельной памятью), взаимного расположения ветвей (на одном процессоре / на разных) и API операционной системы. Параллельное приложение состоит из нескольких ветвей, или процессов, или задач, выполняющихся одновременно. Разные процессы могут выполняться как на разных процессорах, так и на одном и том же - для программы это роли не играет, поскольку в обоих случаях механизм обмена данными одинаков. При программировании на MPI программа должна содержать код всех ветвей сразу. MPI-загрузчиком запускается указываемое количество экземпляров программы. Каждый экземпляр определяет свой порядковый номер в запущенном коллективе, и в зависимости от этого номера и размера коллектива выполняет ту или иную ветку алгоритма. Такая модель параллелизма называется *Single program/Multiple data* (SPMD), и является частным случаем модели *Multiple instruction/Multiple data* (MIMD). Каждая ветвь имеет пространство данных, полностью изолированное от других ветвей. Процессы обмениваются друг с другом данными в виде сообщений. Сообщения проходят под идентификаторами, которые позволяют программе и библиотеке связи отличать их друг от друга. Для совместного проведения тех или иных расчетов процессы внутри приложения объединяются в группы. Каждый процесс может узнать у библиотеки связи свой номер внутри группы, и, в зависимости от номера приступает к выполнению соответствующей части расчетов. Количество ветвей фиксировано - в ходе работы порождение новых ветвей невозможно. Если MPI-приложение запускается в сети, запускаемый файл приложения должен быть построен на каждой машине.
В состав MPI входят, как правило, два обязательных компонента: библиотека программирования для языков Си, Си++ и Фортран, загрузчик исполняемых файлов. Кроме того, может присутствовать справочная система, командные файлы для облегчения компиляции/компоновки программ и др. в зависимости от версии.

Коммуникаторы, группы и области связи.

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

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

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

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

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

Обрамляющие функции. Начало и завершение.

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

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

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

Она получает адреса аргументов, стандартно получаемых самой main от операционной системы и хранящих параметры командной строки. В конец командной строки программы MPI-загрузчик **mpirun** добавляет ряд информационных параметров, которые требуются MPI_Init. (Пример №1).

- Аварийное закрытие библиотеки. Вызывается, если пользовательская программа завершается по причине ошибок времени выполнения, связанных с MPI:

```
MPI_Abort( описатель области связи, код ошибки MPI );
```

Вызов MPI_Abort из любой задачи принудительно завершает работу ВСЕХ задач, подсоединенных к заданной области связи. Если указан описатель MPI_COMM_WORLD, будет завершено все приложение (все его задачи) целиком, что, по-видимому, и является наиболее правильным решением. Используйте код ошибки MPI_ERR_OTHER, если не знаете, как охарактеризовать ошибку в классификации MPI.

- Нормальное закрытие библиотеки:

```
MPI_Finalize();
```

Рекомендуется не забывать вписывать эту инструкцию перед возвращением из программы, то есть:

- перед вызовом стандартной функции Си **exit** ;
- перед каждым после MPI_Init оператором **return** в функции main ;
- если функции main назначен тип void, и она не заканчивается оператором return, то MPI_Finalize() следует поставить в конец main.

- Две информационных функции: сообщают размер группы (то есть общее количество задач, подсоединенных к ее области связи) и порядковый номер вызывающей задачи:

```
int size, rank;
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

Использование MPI_Init, MPI_Finalize, MPI_Comm_size и MPI_Comm_rank. – пример:

```
/*
 *  Начало и завершение:
 *      MPI_Init, MPI_Finalize
 *  Определение количества задач в приложениии и своего порядкового номера:
 *      MPI_Comm_size, MPI_Comm_rank
 */
#include <mpi.h>
```

```
#include <stdio.h>

int main( int argc, char **argv )
{
   int size, rank, i;

    /* Инициализируем библиотеку */
   MPI_Init( &argc, &argv );

    /* Узнаем количество задач в запущенном приложении */
   MPI_Comm_size( MPI_COMM_WORLD, &size );

    /* ... и свой собственный номер: от 0 до (size-1) */
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* задача с номером 0 сообщает пользователю размер группы,
     * к которой прикреплена область связи,
     * к которой прикреплен описатель (коммуникатор) MPI_COMM_WORLD,
     * т.е. число процессов в приложении!!
     */
   if( rank==0 )
      printf("Total processes count = %d\n", size );

    /* Каждая задача выводит пользователю свой номер */
   printf("Hello! My rank in MPI_COMM_WORLD = %d\n", rank );

    /* Точка синхронизации, затем задача 0 печатает
     * аргументы командной строки. В командной строке
     * могут быть параметры, добавляемые загрузчиком MPIRUN.
     */
   MPI_Barrier( MPI_COMM_WORLD );
if( rank == 0 )
      for( puts ("Command line of process 0:"),  i=0; i<argc;i++)
          printf( "%d: \"%s\"\n", i, argv[i] );
/* Все задачи завершают выполнение */
   MPI_Finalize();
   return 0;
}
```

Функции пересылки данных.

Хотя с теоретической точки зрения ветвям для организации обмена данными достаточно всего двух операций (прием и передача), на практике все обстоит гораздо сложнее. Одними только коммуникациями "точка-точка" (т.е. такими, в которых ровно один передающий процесс и ровно один принимающий) занимается порядка 40 функций. Пользуясь ими, программист имеет возможность выбрать:

- способ зацепления процессов - в случае неодновременного вызова двумя процессами парных функций приема и передачи могут быть произведены:
- автоматический выбор одного из трех нижеприведенных вариантов;
    - буферизация на передающей стороне - функция передачи заводит временный буфер, копирует в него сообщение и возвращает управление вызвавшему процессу. Содержимое буфера будет передано в фоновом режиме;
    - ожидание на приемной стороне, завершение с кодом ошибки на передающей стороне;
    - ожидание на передающей стороне, завершение с кодом ошибки на приемной стороне.
- способ взаимодействия коммуникационного модуля MPI с вызывающим процессом:
    - блокирующий - управление вызывающему процессу возвращается только после того, как данные приняты или переданы (или скопированы во временный буфер);
    - неблокирующий - управление возвращается немедленно (т.е. процесс блокируется до завершения операции), и фактическая приемопередача происходит в фоне. Функция неблокирующего приема имеет дополнительный параметр типа "квитанция". Процесс не имеет права производить какие-либо действия с буфером сообщения, пока квитанция не будет "погашена";
- персистентный - в отдельные функции выделены:
    - создание "канала" для приема/передачи сообщения,
    - инициация приема/передачи,
    - закрытие канала.

Такой способ эффективен, к примеру, если приемопередача происходит внутри цикла, а создание/закрытие канала вынесены за его границы.

Две простейшие (но и самые медленные) функции - MPI_Recv и MPI_Send - выполняют блокирующую приемопередачу с автоматическим выбором зацепления (кстати сказать, все функции приема совместимы со всеми функциями передачи). Таким образом, MPI - весьма разветвленный инструментарий. То, что в конкурирующих пакетах типа PVM реализовано одним-единственным способом, в MPI может быть сделано несколькими, про которые говорится: способ А прост в использовании, но не очень эффективен; способ Б сложнее, но эффективнее; а способ В сложнее и эффективнее при определенных условиях. Замечание о разветвленности относится и к коллективным коммуникациям (при которых получателей и/или отправителей несколько): в MPI эта категория представлена 9 функциями 5 типов:
broadcast: один-всем,
scatter: один-каждому,
gather: каждый-одному,
allgather: все-каждому,
alltoall: каждый-каждому.

На первый взгляд может показаться, что программисту легче будет в случае необходимости самому написать такую функцию, но при этом он, скорее всего, будет использовать функции типа MPI_Send и MPI_Recv, в то время как имеющиеся в MPI функции оптимизированы - не пользуясь функциями "точка-точка", они напрямую (на что, согласно идеологии MPI, программа пользователя права не имеет) обращаются к разделяемой памяти и семафорам и к TCP/IP при работе в сети. А если такая архитектурно-зависимая оптимизация невозможна, используется оптимизация архитектурно-независимая: передача производится не напрямую от одного ко всем (время передачи линейно зависит от количества ветвей-получателей), а по двоичному дереву (время передачи логарифмически зависит от количества). Как следствие, скорость работы повышается.

Связь "точка-точка". Простейший набор. Пример.

Это самый простой тип связи между задачами: одна ветвь вызывает функцию передачи данных, а другая - функцию приема. В MPI это выглядит, например, так:
Задача 1 передает:

```
int buf[10];
MPI_Send( buf, 5, MPI_INT, 1, 0, MPI_COMM_WORLD );
```

Задача 2 принимает:

```
int buf[10];
MPI_Status status;
MPI_Recv( buf, 10, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
```

Аргументы функций:
Адрес буфера, из которого в задаче 1 берутся, а в задаче 2 помещаются данные. Наборы данных у каждой задачи свои, поэтому, например, используя одно и то же имя массива в нескольких задачах, указываете не одну и ту же область памяти, а разные, никак друг с другом не связанные.
Размер буфера. Задается **не в байтах**, а в количестве ячеек. Для MPI_Send указывает, сколько ячеек требуется передать (в примере передаются 5 чисел). В MPI_Recv означает максимальную емкость приемного буфера. Если фактическая длина пришедшего сообщения меньше - последние ячейки буфера останутся нетронутыми, если больше - произойдет ошибка времени выполнения.
Тип ячейки буфера. MPI_Send и MPI_Recv оперируют массивами однотипных данных. Для описания базовых типов Си в MPI определены константы MPI_INT, MPI_CHAR, MPI_DOUBLE и так далее, имеющие тип MPI_Datatype. Их названия образуются префиксом "MPI_" и именем соответствующего типа (int, char, double, ...), записанным заглавными буквами. Пользователь может "регистрировать" в MPI свои собственные типы данных, например, структуры, после чего MPI сможет обрабатывать их наравне с базовыми.
Номер задачи, с которой происходит обмен данными. Все задачи внутри созданной MPI группы автоматически нумеруются от 0 до (размер группы-1). В примере задача 0 передает задаче 1, задача 1 принимает от задачи 0.
Идентификатор сообщения. Это целое число от 0 до 32767, которое пользователь выбирает сам. Оно служит той же цели, что и, например, расширение файла - задача-приемник:
по идентификатору определяет смысл принятой информации ;
сообщения, пришедшие в неизвестном порядке, может извлекать из общего входного потока в нужном алгоритму порядке.
Хорошим тоном является обозначение идентификаторов символьными именами посредством операторов "#define" или "const int".
Описатель области связи (коммуникатор). Обязан быть одинаковым для MPI_Send и MPI_Recv.
Статус завершения приема. Содержит информацию о принятом сообщении: его идентификатор, номер задачи-передатчика, код завершения и количество фактически пришедших данных.

Коллективные функции.

Под термином "коллективные" в MPI подразумеваются три группы функций:

- точки синхронизации, или барьеры;

- функции коллективного обмена данными (о них уже упоминалось выше);
- функции поддержки распределенных операций.

Коллективная функция одним из аргументов получает описатель области связи (коммуникатор). Вызов коллективной функции является корректным, только если произведен из всех процессов-абонентов соответствующей области связи, и именно с этим коммуникатором в качестве аргумента (хотя для одной области связи может иметься несколько коммуникаторов, подставлять их вместо друг друга нельзя). В этом и заключается коллективность: либо функция вызывается всем коллективом процессов, либо никем; третьего не дано.
Как поступить, если требуется ограничить область действия для коллективной функции только частью присоединенных к коммуникатору задач, или наоборот - расширить область действия? Нужно создавать временную группу/область связи/коммуникатор на базе существующих.

Основные особенности и отличия от коммуникаций типа "точка-точка":

- на прием и/или передачу работают одновременно все задачи-абоненты указываемого коммуникатора;
- коллективная функция выполняет одновременно и прием, и передачу; она имеет большое количество параметров, часть которых нужна для приема, а часть для передачи; в разных задачах та или иная часть игнорируется;
- как правило, значения ВСЕХ параметров (за исключением адресов буферов) должны быть идентичными во всех задачах;
- MPI назначает идентификатор для сообщений автоматически; кроме того, сообщения передаются не по указываемому коммуникатору, а по временному коммуникатору-дупликату; тем самым потоки данных коллективных функций надежно изолируются друг от друга и от потоков, созданных функциями "точка-точка".

MPI_Bcast рассылает содержимое буфера из задачи, имеющей в указанной области связи номер **root**, во все остальные:

```
MPI_Bcast( buf, count, dataType, rootRank, communicator );
```

MPI_Gather ("совок") собирает в приемный буфер задачи root передающие буфера остальных задач.
Векторный вариант "совка" - **MPI_Gatherv** - позволяет задавать разное количество отправляемых данных в разных задачах-отправителях. Соответственно, на приемной стороне задается массив позиций в приемном буфере, по которым следует размещать поступающие данные, и максимальные длины порций данных от всех задач. Оба массива содержат позиции/длины **не** в байтах, а в количестве ячеек типа recvCount.
MPI_Scatter ("разбрызгиватель") : выполняет обратную "совку" операцию - части передающего буфера из задачи root распределяются по приемным буферам всех задач
И ее векторный вариант - **MPI_Scatterv**, рассылающая части неодинаковой длины в приемные буфера неодинаковой длины.
MPI_Allgather аналогична MPI_Gather, но прием осуществляется не в одной задаче, а во ВСЕХ: каждая имеет специфическое содержимое в передающем буфере, и все получают одинаковое содержимое в буфере приемном. Как и в MPI_Gather, приемный буфер последовательно заполняется данными изо всех передающих. Вариант с неодинаковым количеством данных называется **MPI_Allgatherv**.
MPI_Alltoall : каждый процесс нарезает передающий буфер на куски и рассылает куски остальным процессам; каждый процесс получает куски от всех остальных и поочередно размещает их приемном буфере. Это "совок" и "разбрызгиватель" в одном флаконе. Векторный вариант называется **MPI_Alltoallv**.
Коллективные функции несовместимы с "точка-точка": недопустимым, например, является вызов в одной из принимающих широковещательное сообщение задач MPI_Recv вместо MPI_Bcast.

Точки синхронизации, или барьеры.

Этим занимается всего одна функция:

```
int MPI_Barrier( MPI_Comm comm );
```

MPI_Barrier останавливает выполнение вызвавшей ее задачи до тех пор, пока не будет вызвана изо всех остальных задач, подсоединенных к указываемому коммуникатору. Гарантирует, что к выполнению следующей за MPI_Barrier инструкции каждая задача приступит одновременно с остальными.
Это единственная в MPI функция, вызовами которой гарантированно синхронизируется во времени выполнение различных ветвей! Некоторые другие коллективные функции в зависимости от реализации могут обладать, а могут и не обладать свойством одновременно возвращать управление всем ветвям; но для них это свойство является побочным и необязательным - если Вам нужна синхронность, используйте только MPI_Barrier.
Когда может потребоваться синхронизация? Например, синхронизация используется перед аварийным завершением.
Это утверждение непроверено, но: алгоритмическое необходимости в барьерах, как представляется, нет. Параллельный алгоритм для своего описания требует по сравнению с алгоритмом классическим всего лишь двух дополнительных операций - приема и передачи из ветви в ветвь. Точки синхронизации несут чисто технологическую нагрузку вроде той, что описана в предыдущем абзаце.
Иногда случается, что ошибочно работающая программа перестает врать, если ее исходный текст хорошенько нашпиговать барьерами. Однако программа начнет работать медленнее, например:

```
Без барьеров:           0    xxxx....xxxxxxxxxxxxxxxxxxxx
                      1    xxxxxxxxxxxx....xxxxxxxxxxxx
                                 2    xxxxxxxxxxxxxxxxxxxxx....xx
С барьерами:           0    xxxx....xx(xxxxxxxx(||||xxxxxxxx(||xx
                                 1    xxxxxx(||||x....xxxxxxx(xxxxxxxx(||xx
```

```
                                2     xxxxxx(||||xxxxxxxx(||||..xxxxxxxx(xx
                                       ----------------------------- > Время
```

```
Обозначения:
    x   нормальное выполнение
    .   ветвь простаивает - процессорное время отдано под другие цели
    (   вызван MPI_Barrier
    |   MPI_Barrier ждет своего вызова в остальных ветвях
```

Так что "задавить" ошибку барьерами хорошо только в качестве временного решения на период отладки.

Распределенные операции.

Идея проста: в каждой задаче имеется массив. Над нулевыми ячейками всех массивов производится некоторая операция (сложение/произведение/ поиск минимума/максимума и т.д.), над первыми ячейками производится такая же операция и т.д. Четыре функции предназначены для вызова этих операций и отличаются способом размещения результата в задачах.

MPI_Reduce : массив с результатами размещается в задаче с номером **root**:

```
int vector[16];
int resultVector[16];
MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
for( i=0; i<16; i++ )
    vector[i] = myRank*100 + i;
         MPI_Reduce(
    vector,             /* каждая задача в коммуникаторе предоставляет вектор */
    resultVector,   /* задача номер 'root' собирает данные сюда */
    16,                  /* количество ячеек в исходном и результирующем массивах */
    MPI_INT,      /* и тип ячеек */
    MPI_SUM,    /* описатель операции: поэлементное сложение векторов */
    0,                   /* номер задачи, собирающей результаты в 'resultVector' */
    MPI_COMM_WORLD  /* описатель области связи */
);
if( myRank==0 )
    /* печатаем resultVector, равный сумме векторов */
```

Предопределенных описателей операций в MPI насчитывается 12:

1. **MPI_MAX** и **MPI_MIN** ищут поэлементные максимум и минимум;
2. **MPI_SUM** вычисляет сумму векторов;
3. **MPI_PROD** вычисляет поэлементное произведение векторов;
4. **MPI_LAND**, **MPI_BAND**, **MPI_LOR**, **MPI_BOR**, **MPI_LXOR**, **MPI_BXOR** - логические и двоичные операции И, ИЛИ, исключающее ИЛИ;
5. **MPI_MAXLOC**, **MPI_MINLOC** - поиск индексированного минимума/максимума.

Количество поддерживаемых операциями типов для ячеек векторов строго ограничено вышеперечисленными. Никакие другие встроенные или пользовательские описатели типов использоваться не могут! Все операции являются ассоциативными ("(a+b)+c = a+(b+c)") и коммутативными ("a+b = b+a").

MPI_Allreduce : результат рассылается всем задачам, параметр 'root' убран.

MPI_Reduce_scatter : каждая задача получает не весь массив-результат, а его часть. Длины этих частей находятся в массиве-третьем параметре функции. Размер исходных массивов во всех задачах одинаков и равен сумме длин результирующих массивов.

MPI_Scan : аналогична функции MPI_Allreduce в том отношении, что каждая задача получает результрующий массив. Главное отличие: здесь содержимое массива-результата в задаче **i** является результатом выполнение операции над массивами из задач с номерами от **0** до **i** включительно.

Помимо встроенных, пользователь может вводить свои собственные операции.

Создание коммуникаторов и групп.

Копирование. Самый простой способ создания коммуникатора - скопировать "один-в-один" уже имеющийся:

```
    MPI_Comm tempComm;
    MPI_Comm_dup( MPI_COMM_WORLD, &tempComm );
     /* ... передаем данные через tempComm ... */
    MPI_Comm_free( &tempComm );
```

Новая группа при этом не создается - набор задач остается прежним. Новый коммуникатор наследует все свойства копируемого.

Расщепление. Соответствующая коммуникатору группа расщепляется на непересекающиеся подгруппы, для каждой из которых заводится свой коммуникатор.

```
  MPI_Comm_split(
```

```
    existingComm,                  /* существующий описатель, например MPI_COMM_WORLD */
    indexOfNewSubComm,    /* номер подгруппы, куда надо поместить ветвь */
    rankInNewSubComm,      /* желательный номер в новой подгруппе */
    &newSubComm );            /* описатель области связи новой подгруппы */
```

Эта функция имеет одинаковый первый параметр во всех ветвях, но разные второй и третий - и в зависимости от них разные ветви определяются в разные подгруппы; возвращаемый в четвертом параметре описатель будет принимать в разных ветвях разные значения (всего столько разных значений, сколько создано подгрупп). Если indexOfNewSubComm равен **MPI_UNDEFINED**, то в newSubComm вернется MPI_COMM_NULL, то есть ветвь не будет включена ни в какую из созданных групп.

Создание через группы. В предыдущих двух случаях коммуникатор создается от существующего коммуникатора напрямую, без явного создания группы: группа либо та же самая, либо создается автоматически. Самый же общий способ таков:

функцией **MPI_Comm_group** определяется группа, на которую указывает соответствующий коммуникатор;

на базе существующих групп функциями семейства **MPI_Group_xxx** создаются новые группы с нужным набором ветвей;

для итоговой группы функцией **MPI_Comm_create** создается коммуникатор. Она должна быть вызвана во **всех** ветвях-абонентах коммуникатора, передаваемого первым параметром;

все описатели созданных групп очищаются вызовами функции **MPI_Group_free**.

Такой механизм позволяет не только расщеплять группы подобно MPI_Comm_split, но и объединять их. Всего в MPI определено 7 разных функций конструирования групп.

MPI и типы данных.

О типах передаваемых данных MPI должен знать постольку-поскольку при работе в сетях на разных машинах данные могут иметь разную разрядность (например, тип *int* - 4 или 8 байт), ориентацию (младший байт располагается в ОЗУ первым на процессорах Intel, последним - на всех остальных), и представление (это, в первую очередь, относится к размерам мантиссы и экспоненты для вещественных чисел). Поэтому все функции приемопередачи в MPI оперируют не количеством передаваемых байт, а количеством ячеек, тип которых задается параметром функции, следующим за количеством: MPI_INTEGER, MPI_REAL и т.д. Это переменные типа MPI_Datatype (тип "описатель типов", каждая его переменная описывает для MPI один тип). Они имеются для каждого базового типа, имеющегося в используемом языке программирования.

Однако, пользуясь базовыми описателями, можно передавать либо массивы, либо одиночные ячейки (как частный случай массива). А как передавать данные агрегатных типов, например, структуры? В MPI имеется механизм конструирования пользовательских описателей на базе уже имеющихся (как пользовательских, так и встроенных).

Более того, разработчики MPI создали механизм конструирования новых типов даже более универсальный, чем имеющийся в языке программирования. Действительно, во всех языках программирования ячейки внутри агрегатного типа (массива или структуры):

не налезают друг на друга,

не располагаются с разрывами (выравнивание полей в структурах не в счет).

В MPI сняты оба этих ограничения. Это позволяет весьма причудливо "вырезать", в частности, фрагменты матриц для передачи, и размещать принимаемые данные между собственных.

Выигрыш от использования механизма конструирования типов очевиден - лучше один раз вызвать функцию приемопередачи со сложным шаблоном, чем двадцать раз - с простыми.

Зачем MPI знать тип передаваемых данных?

Действительно, зачем? Стандартные функции пересылки данных прекрасно обходятся без подобной информации - им требуется знать только размер в байтах. Вместо одного такого аргумента функции MPI получают два: количество элементов некоторого типа и символический описатель указанного типа (MPI_INT, и т.д.). Причин тому несколько:

- Пользователю MPI позволяет описывать свои собственные типы данных, которые располагаются в памяти не непрерывно, а с разрывами, или наоборот, с "налезаниями" друг на друга. Переменная такого типа характеризуется не только размером, и эти характеристики MPI хранит в описателе типа.
- Приложение MPI может работать на гетерогенном вычислительном комплексе (коллективе ЭВМ с разной архитектурой). Одни и те же типы данных на разных машинах могут иметь разное представление, например: на плавающую арифметику существует 3 разных стандарта (IEEE,IBM,Cray); тип char в терминальных приложениях Windows представлен альтернативной кодировкой ГОСТ, а в Юниксе - кодировкой KOI-8r ; ориентация байтов в многобайтовых числах на ЭВМ с процессорами Intel отличается от общепринятой (у Intel - младший байт занимает младший адрес, у всех остальных - наоборот). Если приложение работает в гетерогенной сети, через сеть задачи обмениваются данными в формате XDR (eXternal Data Representation), принятом в Internet. Перед отправкой и после приема данных задача конвертирует их в/из формата XDR. Естественно, при этом MPI должен знать не просто количество передаваемых байт, но и тип содержимого.
- Обязательным требованием к MPI была поддержка языка Фортран в силу его инерционной популярности. Фортрановский тип CHARACTER требует особого обращения, поскольку переменная такого типа содержит не собственно текст, а адрес текста и его длину. Функция MPI, получив адрес переменной, должна извлечь из нее адрес текста и копировать сам текст. Это и произойдет, если в поле аргумента-описателя типа стоит MPI_CHARACTER.

Ошибка в указании типа приведет: при отправке - к копированию служебных данных вместо текста, при приеме - к записи текста на место служебных данных. И то, и другое приводит к ошибкам времени выполнения.

- Такие часто используемые в Си типы данных, как структуры, могут содержать в себе некоторое пустое пространство, чтобы все поля в переменной такого типа размещались по адресам, кратным некоторому четному числу (часто 2, 4 или 8) - это ускоряет обращение к ним. Причины тому чисто аппаратные. Выравнивание данных настраивается ключами компилятора. Разные задачи одного и того же приложения, выполняющиеся на одной и той же машине (даже на одном и том же процессоре), могут быть построены с разным выравниванием, и типы с одинаковым текстовым описанием будут иметь разное двоичное представление. MPI будет вынужден позаботиться о правильном преобразовании. Например, переменные такого типа могут занимать 9 или 16 байт:

```
typedef struct {
    char    c;
    double  d;
} CharDouble;
```

Использование MPI.

MPI сам по себе является средством:

- сложным: спецификация на MPI-1 содержит 300 страниц, на MPI-2 - еще 500 (причем это *только отличия и добавления* к MPI-1), и программисту для эффективной работы так или иначе придется с ними ознакомиться, помнить о наличии нескольких сотен функций, и о тонкостях их применения;
- специализированным: это система связи.

Можно сказать, что сложность (т.е. многочисленность функций и обилие аргументов у большинства из них) является ценой за компромисс между эффективностью и универсальностью. С одной стороны, на SMP-машине должны существовать способы получить *почти* столь же высокую скорость при обмене данными между ветвями, как и при традиционном программировании через разделяемую память и семафоры. С другой стороны, все функции должны работать на любой платформе. Таким образом, программист заинтересован в инструментах, которые облегчали бы:

проведение декомпозиции,
запись ее в терминах MPI.

В данном случае это средства, генерирующие на базе входных данных текст программы на стандартном Си или Фортране, обладающей явным параллелизмом, выраженным в терминах MPI; содержащий вызовы MPI-процедур, наиболее эффективные в окружающем контексте. Такие средства делают написание программы не только легче, но и надежнее. Назовем некоторые перспективные типы такого инструментария, который лишал бы программиста необходимости вообще помнить о присутствии MPI.

Средства автоматической декомпозиции. Идеалом является такое оптимизирующее средство, которое на входе получает исходный текст некоего последовательного алгоритма, написанный на обычном языке программирования, и выдает на выходе исходный текст этого же алгоритма на этом же языке, но уже в распараллеленном на ветви виде, с вызовами MPI. Что ж, такие средства созданы, но никто не торопится раздавать их бесплатно. Кроме того, вызывает сомнение их эффективность.

Языки программирования. Это наиболее популярные на сегодняшний день средства полуавтоматической декомпозиции. В синтаксис универсального языка программирования (Си или Фортрана) вводятся дополнения для записи параллельных конструкций кода и данных. Препроцессор переводит текст в текст на стандартном языке с вызовами MPI. Примеры таких систем: mpC (massively parallel C) и HPF (High Performance Fortran).

Оптимизированные библиотеки для стандартных языков. В этом случае оптимизация вообще может быть скрыта от проблемного программиста. Чем больший объем работы внутри программы отводится подпрограммам такой библиотеки, тем большим будет итоговый выигрыш в скорости ее (программы) работы. Собственно же программа пишется на обычном языке программирования безо всяких упоминаний об MPI, и строится стандартным компилятором. От программиста потребуется лишь указать для компоновки имя библиотечного файла MPI, и запускать полученный в итоге исполняемый код не непосредственно, а через MPI-загрузчик. Популярные библиотеки обработки матриц, такие как Linpack, Lapack и ScaLapack, уже переписаны под MPI.

Средства визуального проектирования. Действительно, почему бы не расположить на экране несколько окон с исходным текстом ветвей, и пусть пользователь легким движением мыши протягивает стрелки от точек передачи к точкам приема - а визуальный построитель генерирует полный исходный текст?

Отладчики. Об отладчиках пока можно сказать только то, что они нужны. Должна быть возможность одновременной трассировки/просмотра нескольких параллельно работающих ветвей - что-либо более конкретное пока сказать трудно.

MPI-1 и MPI-2.

В функциональности MPI есть пробелы, которые устранены в следующем проекте, MPI-2. Вкратце перечислим наиболее важные нововведения:

- Взаимодействие между приложениями. Поддержка механизма "клиент-сервер".
- Динамическое порождение ветвей.
- Для работы с файлами создан архитектурно-независимый интерфейс.
- Сделан шаг в сторону SMP-архитектуры. Теперь разделяемая память может быть не только каналом связи между ветвями, но и местом совместного хранения данных.

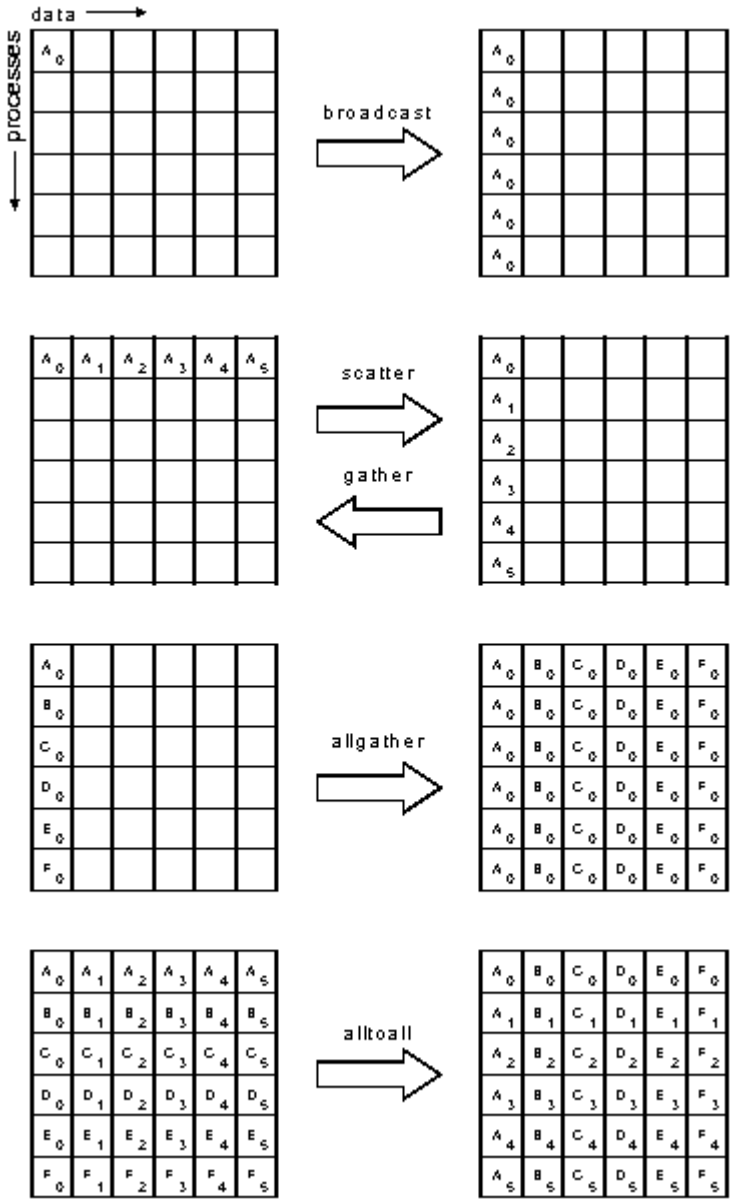
Пример.

```
/*
 *  Простейшая приемопередача:
 *     MPI_Send, MPI_Recv
 *  Завершение по ошибке:
 *     MPI_Abort
 */

#include <mpi.h>
#include <stdio.h>

  /* Идентификаторы сообщений */
#define tagFloatData  1
#define tagDoubleData 2

  /* Этот макрос введен для удобства, */
  /* он позволяет указывать длину массива в количестве ячеек */
#define ELEMS(x)   ( sizeof(x) / sizeof(x[0]) )

int main( int argc, char **argv )
{
    int size, rank, count;
    float floatData[10];
    double doubleData[20];
    MPI_Status status;
      /* Инициализируем библиотеку */
    MPI_Init( &argc, &argv );
      /* Узнаем количество задач в запущенном приложении */
    MPI_Comm_size( MPI_COMM_WORLD, &size );
      /* ... и свой собственный номер: от 0 до (size-1) */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
      /* пользователь должен запустить ровно две задачи, иначе ошибка */
    if( size != 2 ) {
          /* задача с номером 0 сообщает пользователю об ошибке */
        if( rank==0 )
            printf("Error: two processes required instead of %d, abort\n",
                size );
          /* Все задачи-абоненты области связи MPI_COMM_WORLD
           * будут стоять, пока задача 0 не выведет сообщение.
           */
        MPI_Barrier( MPI_COMM_WORLD );
          /* Без точки синхронизации может оказаться, что одна из задач
           * вызовет MPI_Abort раньше, чем успеет отработать printf()
           * в задаче 0, MPI_Abort немедленно принудительно завершит
           * все задачи и сообщение выведено не будет
           */
          /* все задачи аварийно завершают работу */
        MPI_Abort(
            MPI_COMM_WORLD,  /* Описатель области связи, на которую */
                             /*    распространяется действие ошибки */
            MPI_ERR_OTHER ); /* Целочисленный код ошибки */
        return -1;
    }
    if( rank==0 ) {
      /* Задача 0 что-то такое передает задаче 1 */
        MPI_Send(
            floatData,        /* 1) адрес передаваемого массива */
            5,       /* 2) сколько: 5 ячеек, т.е. floatData[0]..floatData[4] */
```

```
            MPI_FLOAT,        /* 3) тип ячеек */
            1,                /* 4) кому: задаче 1 */
            tagFloatData,     /* 5) идентификатор сообщения */
            MPI_COMM_WORLD ); /* 6) описатель области связи, через которую */
                              /*    происходит передача */
          /* и еще одна передача: данные другого типа */
        MPI_Send( doubleData, 6, MPI_DOUBLE, 1, tagDoubleData, MPI_COMM_WORLD );
    } else {
      /* Задача 1 что-то такое принимает от задачи 0 */
        /* дожидаемся сообщения и помещаем пришедшие данные в буфер */
        MPI_Recv(
            doubleData,     /* 1) адрес массива, куда складывать принятое */
            ELEMS( doubleData ), /* 2) фактическая длина приемного */
                               /*    массива в числе ячеек */
            MPI_DOUBLE,     /* 3) сообщаем MPI, что пришедшее сообщение */
                            /*    состоит из чисел типа 'double' */
            0,              /* 4) от кого: от задачи 0 */
            tagDoubleData,  /* 5) ожидаем сообщение с таким идентификатором */
            MPI_COMM_WORLD, /* 6) описатель области связи, через которую */
                            /*    ожидается приход сообщения */
            &status );      /* 7) сюда будет записан статус завершения приема */

          /* Вычисляем фактически принятое количество данных */
        MPI_Get_count(
            &status,        /* статус завершения */
            MPI_DOUBLE,     /* сообщаем MPI, что пришедшее сообщение */
                            /*    состоит из чисел типа 'double' */
            &count );       /* сюда будет записан результат */

          /* Выводим фактическую длину принятого на экран */
        printf("Received %d elems\n", count );

          /* Аналогично принимаем сообщение с данными типа float
           * Обратите внимание: задача-приемник имеет возможность
           * принимать сообщения не в том порядке, в котором они
           * отправлялись, если эти сообщения имеют разные идентификаторы
           */
        MPI_Recv( floatData, ELEMS( floatData ), MPI_FLOAT,
            0, tagFloatData, MPI_COMM_WORLD, &status );
        MPI_Get_count( &status, MPI_FLOAT, &count );
    }
      /* Обе задачи завершают выполнение */
    MPI_Finalize();
    return 0;
}
```

Приложение.

Коллективные функции:



Общие функции

int MPI_Init(int* argc, char* argv)**
int MPI_Finalize(void)

int MPI_Comm_size(MPI_Comm comm, int* size)
int MPI_Comm_rank(MPI_comm comm, int* rank)

double MPI_Wtime(void)

int MPI_Barrier(MPI_Comm comm)

16. Методы представления знаний в системах искусственного интеллекта (язык предикатов, семантические сети, фреймы, продукции).

Составляющие обучения:

1) Знания - усвоенные Понятия
2) Умения - автоматизированные действия по решению определенных задач
3) Навыки - адаптивные способности человека

Все три составляющие представляют знания в ИИ. Знания хранятся в Базе Знаний (БЗ).

База знаний состоит из:

1) Базы фактов
2) Базы правил
3) Базы понятий
4) Базы процедур (то есть умений)
5) Базы целей
6) Базы метазнаний (то есть знаний о себе)
7)

Способы представления знаний:
1) декларативные
2) процедурные

Метод представления знаний - совокупность взаимосвязанных средств описания знаний и оперирования этими описаниями.

Методы представления.

Логические методы (язык предикатов)
Знания, необходимые для решения задач и организации взаимодействия с пользователем, - факты (утверждения). Факт - формула в некоторой логике. Система знаний - совокупность формул. База знаний - система знаний в компьютерном представлении.

Основные операции: логический вывод (доказательство теорем).

Достоинства:
1) формальный аппарат вывода (новых фактов/знаний из известных фактов/знаний)
2) возможность контроля целостности
3) простая и ясная нотация.

Недостатки:
1) знания трудно структурировать
2) при большом количестве формул вывод идет очень долго
3) при большом количестве формул их совокупность трудно обозрима.

Семантические сети
Знания, необходимые для решения задач и организации взаимодействия с пользователем, - объекты/события и связи между ними. Статические семантические сети - сети с объектами. Динамические семантические сети (сценарии) - сети с событиями. Система знаний - совокупность сетей (или одна общая сеть). База знаний - система знаний в компьютерном представлении.

Для представления семантических сетей используются графы:
1) вершина - атомарный объект (событие),
2) подграф - структурно сложный объект (событие),
3) дуга - отношение или действие

Основные операции: сопоставление с образцом, поиск, замена, взятие копии.

Достоинства: знания хорошо структурированы, структура понятна человеку.

Недостатки:
1) при большом объеме сети очень долго выполняются все операции;
2) при большом объеме сети она трудно обозрима.

Фреймы
Знания, необходимые для решения задач и организации взаимодействия с пользователем, - фреймы. Фрейм-понятие - отношение/действие + связанные этим отношением/участвующие в этом действии объекты. Фрейм-пример - конкретный экземпляр отношения/действия + конкретные объекты (связанные этим отношением/участвующие в этом действии). Система знаний - совокупность фреймов-понятий и фреймов-примеров. База знаний - система знаний в компьютерном представлении.

Фрейм:
1) ИМЯ - отношение/действие
2) СЛОТЫ - объекты или другие фреймы. С каждым слотом может быть связана такая информация:
    a) УСЛОВИЕ НА ЗАПОЛНЕНИЕ (тип, "по умолчанию", связь с другими слотами)
    b) АССОЦИИРОВАННЫЕ ПРОЦЕДУРЫ (действия, выполняемые, например, при заполнении этого слота)

Основные операции: поиск фрейма/слота, замена значения слота, взятие копии фрейма-понятия.

Достоинства: знания хорошо структурированы, структура понятна человеку.

Недостатки:
1) при большом количестве фреймов долго выполняются все операции;
2) при большом количестве фреймов знания трудно обозримы.

Продукции
Знания, необходимые для решения задач и организации взаимодействия с пользователем, - продукции (продукционные правила). Продукция - правило вида: p: a → b (где: p - предусловие, a - антецедент, b - консеквент). Система знаний - система продукционных правил + стратегия выбора правил. База знаний - система знаний в компьютерном представлении.

Основные операции: вывод (применение правила, определение правила-преемника и т.д.).

Достоинства: простая и ясная нотация.

Недостатки:
1) при большом количестве правил вывод идет очень долго;
2) при большом количестве правил их совокупность трудно обозрима.

17. Методы поиска решения задач в системах искусственного интеллекта (эвристический поиск в пространстве состояний и на И/ИЛИ деревьях).

Представление задач в пространстве состояний
Ключевым понятием при формализации задачи в пространстве состояний является понятие состояния, характеризующего некоторый момент решения задачи. Среди всех состояний выделяются начальное состояние и целевое состояние (целевая конфигурация), в совокупности определяющие задачу, которую надо решить.

Другим важным понятием для рассматриваемого представления является понятие оператора, или допустимого хода в задаче. Оператор преобразует одно состояние в другое, являясь, по сути, функцией, определенной на множестве состояний и принимающей значения из этого множества.

В терминах состояний и операторов решение задачи есть определенная последовательность операторов, преобразующая начальное состояние в целевое. Решение задачи ищется в пространстве состояний - множестве состояний, достижимых из начального состояния при помощи операторов.

Пространство состояний можно представить в виде графа, вершины которого соответствуют состояниям, а дуги - применяемым операторам. Тогда решение задачи - это путь, ведущий от начального состояния к целевому. Пространства состояний могут быть большими и даже бесконечными, но в любом случае предполагается конечность множества допустимых операторов и счетность множества возможных состояний.

Итак, формализация задачи с использованием пространства состояний включает выявление и определение следующих составляющих:
1) формы описания состояний и описание исходной задачи;
2) множество операторов и их воздействий на описания состояний;
3) указание свойств целевых состояний (или же явное их задание).

Эти составляющие задают (неявно) пространство, в котором требуется провести поиск решения задачи.
Ясно, что решение задачи, представленной описанным способом, можно в принципе обнаружить, осуществляя последовательный поиск, или перебор вершин, в пространстве состояний. В начале этого процесса к начальному состоянию применяется тот или иной оператор. Затем на каждом шаге поиска к одному из уже полученных (просмотренных) состояний применяется допустимый оператор и строится новая вершина. Поиск заканчивается, когда построено целевое состояние.

Алгоритмы поиска решения (в пространстве состояний)
Алгоритмы поиска в пространстве состояний базируются на последовательном переборе вершин пространства состояния - до тех пор, пока не будет обнаружена целевая вершина.
Вершины и указатели, построенные в процессе перебора, образуют поддерево всего неявно определенного пространства состояний. Будем называть такое поддерево деревом перебора.
Известные алгоритмы поиска в пространстве состояний различаются несколькими характеристиками:
1) использованием или нет эвристической информации (соответственно, слепые и эвристические алгоритмы);
2) порядком раскрытия (обхода) вершин (соответственно в ширину и глубину)
3) полнотой просмотра пространства (просмотр либо полного пространства, либо его части)
4) направлением поиска (прямые – от начальной вершины к целевой, обратные – от целевой к начальной и двунаправленные).

Поиск на игровых деревьях
Будем рассматривать класс игр двух лиц с полной информацией. В таких играх участвуют два игрока, которые поочередно делают свои ходы. В любой момент игры каждому игроку известно все, что произошло в игре к этому моменту и что может быть сделано в настоящий момент. Игра заканчивается либо выигрышем одного игрока (и проигрышем другого), либо ничьей.

Для формализации и изучения игровых стратегий в классе игр с полной информацией может быть использован подход, основанный на редукции задач. Напомним, что при этом должны быть определены следующие составляющие: форма описания задач и подзадач; операторы, сводящие задачи к подзадачам; элементарные задачи; а также задано описание исходной задачи.

Рассмотрим задачу поиска выигрышной стратегии для одного из игроков, отправляясь от некоторой фиксированной конфигурации (позиции) игры (не обязательно начальной). При использовании подхода, основанного на редукции задач, выигрышная стратегия ищется в процессе доказательства того, что игра может быть выиграна. Аналогично, поиск ничейной стратегии, исходя из некоторой конкретной позиции, ведется в процессе доказательства того, что игра может быть сведена к ничьей.

X_S (или Y_S) - некоторая конфигурация игры, причем индекс S принимает значения "+" или "-", указывая тем самым, кому принадлежит следующий ход, то есть в конфигурации X+ следующий ход должен делать первый игрок, а в X- второй.
$W(X_S)$ - задача доказательства того, что первый игрок может выиграть, исходя из конфигурации X_S;
$V(X_S)$ - задача доказательства того, что второй игрок может выиграть, отправляясь от конфигурации X_S.

Рассмотрим игровую задачу $W(X_S)$. Операторы сведения этой задачи к подзадачам определяются исходя из ходов, допустимых в проводимой игре:

1) Если в некоторой конфигурации X+ очередь делать ход за игроком один, и имеется N допустимых ходов, приводящих соответственно к конфигурациям X_1-, X_2-, ..., X_N- , то для решения задачи W(X+) необходимо решить по крайней мере одну из подзадач $W(X_i-)$ (так как ход выбирает первый игрок, то он выиграет игру, если хотя бы один из ходов ведет к выигрышу).
2) Если же в некоторой конфигурации Y- ход должен сделать игрок два, и имеется K допустимых ходов, приводящих к конфигурациям Y_1+, Y_2+, ..., Y_K+, то для решения задачи W(Y-) требуется решить каждую из возникающих подзадач $W(Y_i+)$ (так как ход выбирает второй игрок, то первый выиграет игру, если выигрыш гарантирован ему после любого хода противника).

Последовательное применение для исходной конфигурации игры данной схемы сведения игровых задач к совокупности подзадач порождает И/ИЛИ-дерево (И/ИЛИ-граф), которое называют деревом (графом) игры.
Дуги игрового дерева соответствуют ходам игроков, вершины - конфигурациям игры, причем листья дерева - это позиции, в которых игра завершается выигрышем, проигрышем или ничьей. Часть листьев являются заключительными вершинами, соответствующими элементарным задачам - позициям, выигрышным для первого игрока. Заметим, что для конфигураций, где ход принадлежит первому игроку, в игровом дереве получается ИЛИ-вершина, а для позиций, в которых ходит второй игрок, - И-вершина.

Цель построения игрового дерева или графа - получение решающего поддерева (подграфа) для задачи $W(X_S)$, показывающего, как первый игрок может выиграть игру из позиции X_S независимо от ответов противника.

В случаях, когда пространство ходов велико (как и во всех сложных играх), вместо нереальной задачи поиска полной игровой стратегии решается, как правило, более простая задача - поиск для заданной позиции игры достаточно хорошего первого хода.

Минимаксная процедура
С целью поиска достаточно хорошего первого хода просматривается обычно часть игрового дерева, построенного от заданной конфигурации. Для этого применяется один из переборных алгоритмов (в глубину, в ширину или эвристический) и некоторое искусственное окончание перебора вершин в игровом дереве: например, ограничивается время перебора или же глубина поиска.

После построения таким образом частичного дерева игры вершины в нем оцениваются, и по этим оценкам определяется наилучший ход от заданной игровой конфигурации. При этом:

1) Листовые вершины оцениваются статической оценочной функций
2) Нелистовые вершины оцениваются по минимаксному принципу.

Значение статической оценочной функции тем больше, чем больше преимуществ имеет первый игрок (над вторым игроком) в оцениваемой позиции. Очень часто оценочная функция выбирается следующим образом:

1) статическая оценочная функция положительна в игровых конфигурациях, где первый игрок имеет преимущества;
2) статическая оценочная функция отрицательна в конфигурациях, где второй игрок имеет преимущества;
3) статическая оценочная функция близка к нулю в позициях, не дающих преимущества ни одному из игроков.

Минимаксный принцип:

1) ИЛИ-вершине дерева игры приписывается оценка, равная максимуму оценок ее дочерних вершин;
2) И-вершине игрового дерева приписывается оценка, равная минимуму оценок ее дочерних вершин.

Основные этапы минимаксной процедуры:

1) Дерево игры строится (просматривается) одним из известных алгоритмов.
2) Все концевые вершины полученного дерева, то есть вершины, находящиеся на глубине N, оцениваются с помощью статической оценочной функции.
3) В соответствии с минимаксным принципом вычисляются оценки всех остальных вершин.
4) Среди вершин, дочерних к начальной, выбирается вершина с наибольшей оценкой: ход, который к ней ведет, и есть искомый наилучший ход в игровой конфигурации S.

Альфа-бета процедура

Правила вычисления оценок вершин дерева игры, в том числе предварительных оценок промежуточных вершин, которые для удобства будем называть α и β -величинами:

1) концевая вершина дерева оценивается статической оценочной функцией сразу, как только она построена;
2) промежуточная вершина предварительно оценивается по минимаксному принципу, как только стала известна оценка хотя бы одной из ее дочерних вершин; каждая предварительная оценка пересчитывается (уточняется) всякий раз, когда получена оценка еще одной дочерней вершины;
3) предварительная оценка ИЛИ-вершины (α -величина) полагается равной наибольшей из вычисленных к текущему моменту оценок ее дочерних вершин;
4) предварительная оценка И-вершины (β -величина) полагается равной наименьшей из вычисленных к текущему моменту оценок ее дочерних вершин.

Укажем очевидное следствие этих правил вычисления: α -величины не могут уменьшаться, а β -величины не могут увеличиваться.

Сформулируем теперь правила прерывания перебора, или отсечения ветвей игрового дерева:

1) α -отсечение: Перебор можно прервать ниже любой И-вершины, β -величина которой не больше, чем α - величина одной из предшествующих ей ИЛИ-вершин (включая корневую вершину дерева);
2) β -отсечение: Перебор можно прервать ниже любой ИЛИ-вершины, α -величина которой не меньше, чем β - величина одной из предшествующих ей И-вершин.

Утверждение: α - β процедура всегда приводит к тому же результату (наилучшему первому ходу), что и простая минимаксная процедура той же глубины.

18. Организация сетевого взаимодействия. Эталонная модель OSI ISO. Основные элементы и архитектура OSI ISO. Уровни протоколов и их основные функции.

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

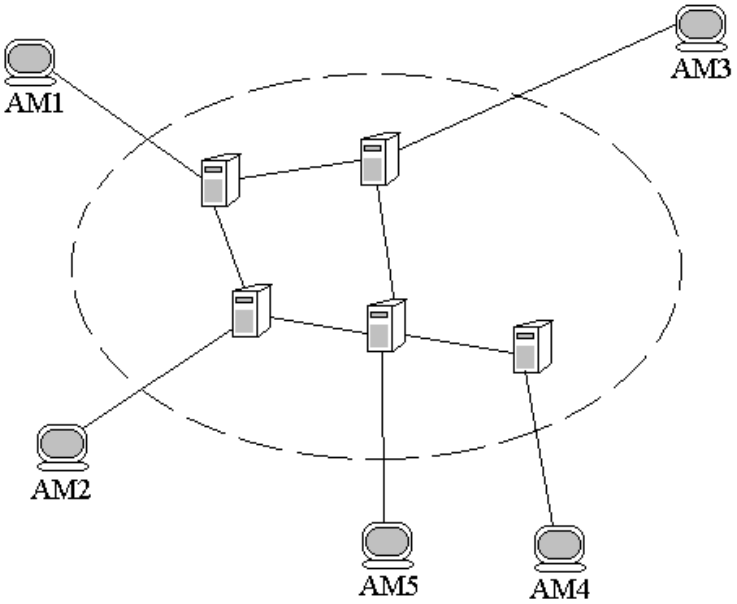
- сеть может состоять из значительного числа взаимодействующих друг с другом ЭВМ, обеспечивающих сбор, хранение, обработку и передачу информации;
- сеть ЭВМ предполагает возможность распределенной обработки информации;
- расширяемость сети – возможность развития сети по протяженности (размещению), по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

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

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

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

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



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

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

Сеть коммутации сообщений. Если коммутация каналов - это коммутация на время всего сеанса связи, то коммутация сообщений - это связь, при которой весь сеанс разделяется на передачу сообщений (сообщение - некоторая, логически завершенная, порция данных). Взаимодействие ЭВМ в данных сетях осуществляется в терминах передачи сообщений. При этом для передачи сообщения не требуется установления прямого соединения между взаимодействующими абонентскими ЭВМ. Сообщение начинает передаваться по сети по мере освобождения каналов в коммуникационной среде (***абонентская_ЭВМ⇔коммуникационная_ЭВМ***; ***коммуникационная_ЭВМ⇔коммуникационная_ЭВМ***). При этом на коммуникационные машины ложится значительная дополнительная нагрузка – коммуникационные ЭВМ должны обеспечивать буферизацию сообщений, так как прямого соединения не устанавливается, и возможна ситуация при которой канал, необходимый для продолжения передачи сообщений, занят. Достоинства - логическая и физическая простота, снижение монопольного фактора по сравнению с сетью коммутации каналов. Недостатки - снижение скорости работы в сети, отсутствие детерминированности в определении времени доставки информации, необходимость существенного увеличения мощности коммуникационных ЭВМ для обеспечения буферизации.

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

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

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

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

был стандартизировать основные интерфейсы, позволяющие строить и развивать сети ЭВМ. Была предложена семиуровневая модель организации взаимодействия в сети.

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

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

Модель OSI служит функциональным руководством при решении связных задач, и поэтому не специфицирует каких-либо коммуникационных стандартов, позволяющих решать эти задачи. Однако многие коммуникационные стандарты и протоколы вполне согласуются с положениями Модели OSI.

В модели OSI центральными являются три понятия:

1) сервис - определяет что делает уровень, но ничего не говорит как
2) интерфейс - определяет для вышележащего уровня доступа к сервису
3) протокол - определяет реализацию сервиса

Наибольшее методологическое значение этой модели в четком выделении и разделении этих понятий.

В Модели OSI применяется стратегия "разделяй и властвуй". Каждый уровень выполняет определенные функции. Уровни и их функции были выбраны на основе естественного разделения на подзадачи. Каждый уровень на одной ЭВМ связывается с аналогичным уровнем другой ЭВМ, однако данная связь реализуется в результате передачи сообщений через соответствующие нижележащие уровни. При этом межуровневая связь четко определяется. Уровень N использует услуги уровня N-1, обеспечивая услуги для уровня N+1.

В каждом запросе имеется так называемый заголовок, содержащий управляющую информацию. Любой уровень может добавлять заголовок к сообщению. На каждом уровне сообщение представляется в виде двух частей: заголовок и данные. Важно понять, что эти термины являются относительными. Когда уровень 4 добавляет свой заголовок и передает сообщение на уровень 3, третий уровень может добавить свой собственный заголовок к тому, что получено от уровня 4. При этом "данные" уровня 3 включают заголовок и данные уровня 4.

Добавление заголовков является необходимым, но при этом происходит добавление довольно большого количества информации даже к очень коротким сообщениям. Например, к моменту достижения 15-ти символьным почтовым сообщением среды передачи данных его длина может увеличиться в 5 раз. Исходное почтовое сообщение и его заголовки передаются по сети в устройство назначения. ЭВМ назначения отделяет и обрабатывает заголовки в обратном порядке. В конце концов пользователь получит исходное почтовое сообщение.

Информационные блоки именуются по-разному в зависимости от обсуждаемого уровня Модели. На физическом уровне мы говорим о битах. На звеньевом уровне логические группы информации называются кадрами. На сетевом уровне часто - дейтаграммой. На транспортном уровне те же базовые элементы данных называются сегментами. На прикладном уровне элементы данных обычно называются сообщениями. Другие термины (включая пакет) также применяются на различных уровнях.

Физический уровень
Физический уровень отвечает за передачу последовательности битов через канал связи. Основной проблемой является как гарантировать что если на одном конце послали 1, то на другом получили 1, а не 0. На этом уровне решают такие вопросы, как: каким напряжением надо представлять 1, а каким - 0; сколько микросекунд тратиться на передачу одного бита; следует ли поддерживать передачу данных в обоих направлениях одновременно; как устанавливается начальное соединение и как оно разрывается; каково количество контактов на сетевом разъеме, для чего используется каждый контакт. Здесь в основном вопросы механики, электрики.

Уровень канала данных
Основной задачей уровня канала данных - превратить несовершенную среду передачи в надежный канал, свободный от ошибок передачи. Эта задача решается разбиением данных отправителя на фреймы (обычно от нескольких сотен до нескольких тысяч байтов), передачей фреймов последовательно и обработкой фреймов уведомления, поступающих от получателя. Поскольку физический уровень не распознает структуры в передаваемых данных, то это целиком и полностью задача канала данных определить границы фрейма. Эта задача решается введением специальной последовательности битов, которая добавляется в начало и в конец фрейма и всегда интерпретируется как границы фрейма.

Помехи на линии могут разрушить фрейм. В этом случае он должен быть передан повторно. Он будет повторен также и в том случае если фрейм уведомление будет потерян. И это уже заботы уровня как бороться с дубликатами одного и того же фрейма, потерями или искажениями фреймов. Уровень канала данных может поддерживать сервис разных классов для сетевого уровня, разного качества и стоимости.

Другой проблемой, возникающей на уровне канала данных (равно как и на других вышележащих уровнях) как управлять потоком передачи. Например, как предотвратить "захлебывание" получателя. Как сообщить передающему размер буфера, для приема передаваемых данных имеющийся у получателя в этот момент.

Если канал позволяет передавать данные в обоих направлениях одновременно, то возникает новая проблема: фреймы уведомления для потока от А к В используют тот же канал, что и трафик от В к А. Решение - использовать фреймы DU для передачи фреймов уведомлений.

В сетях с вещательным способом передачи возникает проблема управления доступом к общему каналу. За это отвечает специальный подуровень - подуровень доступа к среде (MAC - Media ACcess).

Сетевой уровень
Сетевой уровень отвечает за функционирование подсети. Основной проблемой здесь является то, как маршрутизировать пакеты от отправителя к получателю. Маршруты могут быть определены заранее и прописаны в статической таблице, которая не изменяется. Они могут определяться в момент установления соединения. Наконец, они могут строиться динамически в зависимости от загрузки сети.

Если в подсети циркулирует слишком много пакетов, то они могут использовать одни и те же маршруты, что будет приводить к заторам. Эта проблема так же решается на сетевом уровне.

Поскольку за использование подсети, как правило, предполагается оплата, то на этом уровне также присутствуют функции учета: как много байт, символов послал или получил абонент сети. Если абоненты расположены в разных странах, где разные тарифы, то надо должным образом скорректировать цену услуги.

Если пакет адресован в другую сеть, то надо предпринять надлежащие меры: там может быть другой формат пакетов, отличный способ адресации, размер пакетов, протоколы и т.д. - это все проблемы неоднородных сетей решаются на сетевом уровне.

В сетях с вещательной передачей проблемы маршрутизации просты и этот уровень часто отсутствует.

Транспортный уровень
Основная функция транспортного уровня это: принять данные с уровня сессии, разделить, если надо, на более мелкие единицы, передать на сетевой уровень и позаботиться, чтобы все они дошли в целостности до адресата. Все это должно быть сделано эффективно и так, чтобы скрыть от вышележащего уровня непринципиальные изменения на нижних.

В нормальных условиях транспортный уровень должен создать специальное сетевое соединение для каждого транспортного соединения по запросу уровня сессии. Если транспортное соединение требует высокой пропускной способности, то транспортный уровень может создать несколько сетевых соединений, между которыми транспортный уровень буден

распределять передаваемые данные. И наоборот, если требуется обеспечить недорогое транспортное соединение, то транспортный уровень может использовать одно и то же сетевое соединение для нескольких транспортных соединений. В любом случае, такое мультиплексирование должно быть незаметным на уровне сессии.

Сетевой уровень определяет, какой тип сервиса предоставить вышележащим уровням и пользователям сети. Наиболее часто используемым сервисом является канал точка-точка без ошибок, обеспечивающий доставку сообщений или байтов в той последовательности, в какой они были отправлены. Другой вид сервиса - доставка отдельных сообщений без гарантии сохранения их последовательности, рассылка одного сообщения многим в режиме вещания. Тип сервиса определяется при установлении транспортного соединения.

Транспортный уровень - это действительно уровень, обеспечивающий соединение точка-точка. Активности транспортного уровня на машине отправителя общаются с равнозначными активностями транспортного уровня на машине получателя. Этого нельзя сказать про активности на нижележащих уровнях. Они общаются с равнозначными активностями на соседних машинах! В этом одно из основных отличий уровней 1-3 от уровней 4-7. Последние обеспечивают соединение точка-точка.

Многие хост-машины - мультипрограммные, поэтому транспортный уровень для одной такой машины должен поддерживать несколько транспортных соединений. Для того, чтобы определить к какому соединению относиться тот или иной пакет, в его заголовке помещается необходимая информация.

Транспортный уровень также отвечает за установление и разрыв транспортного соединения в сети. Это предполагает наличие механизма именования, т.е. процесс на одной машине должен уметь указать с кем в сети ему надо обменяться информацией. Транспортный уровень также должен предотвращать "захлебывание" получателя в случае очень "быстро говорящего" отправителя. Механизм для этого называется управление потоком. Он есть и на других уровнях. Однако, управление потоком между хостами отличен от управление потоком между маршрутизаторами.

Уровень сессии
Уровень сессии позволяет пользователям на разных машинах (напомним, что пользователем может быть программа) устанавливать сессии. Сессия позволяет передавать данные, как это может делать транспортный уровень, но кроме этого этот уровень имеет более сложный сервис, полезный в некоторых приложениях. Например, вход в удаленную систему, передать файл между двумя приложениями.

Одним из видов услуг на этом уровне - управление диалогом. Потоки данных могут быть разрешены в обоих направлениях одновременно, либо поочередно в одном направлении. Сервис на уровне сессии будет управлять направлением передачи.

Другим видом сервиса - управление маркером. Для некоторых протоколов недопустимо выполнение одной и той же операции на обоих концах соединения одновременно. Для этого уровень сессии выделяет активной стороне маркер. Операцию может выполнять тот, кто владеет маркером.

Другой услугой уровня сессии является синхронизация. Пусть нам надо передать файл такой, что его пересылка займет два часа, между машинами, время наработки на отказ у которых один час. Ясно что "в лоб" такой файл средствами транспортного уровня не решить. Уровень сессии позволяет расставлять контрольные точки. В случае отказа одной из машин передача возобновиться с последней контрольной точки.

Уровень представления
Уровень представления предоставляет решения для часто возникающих проблем, чем облегчает участь пользователей. В основном это проблемы семантики и синтаксиса передаваемой информации. Этот уровень имеет дело с информацией, а не с потоком битов.

Типичным примером услуги на этом уровне - унифицированная кодировка данных. Дело в том, что на разных машинах используются разные способы кодировки ASCII, Unicode и т.п. для символов, разные способы представления целых - в прямом, обратном или дополнительном коде, нумерация бит в байте слева направо или наоборот и т.п. Пользователи как правило используют структуры данных, а не случайный набор байт. Для того, чтобы машины с разной кодировкой и представлением данных могли взаимодействовать, передаваемые структуры данных определяются специальным абстрактным способом, не зависящим от кодировки, используемой при передачи. Уровень представления работает со структурами данных в абстрактной форме, преобразует это представление во внутреннее для конкретной машины и из внутреннего, машинного представления в стандартное представление для передачи по сети.

Уровень приложений
Уровень приложений обеспечивает нужные часто используемые протоколы. Например, существуют сотни разных типов терминалов. Если мы захотим создать сетевой экранный редактор, то нам придется писать для каждого типа терминала свою версию.

Есть другой путь: определить сетевой виртуальный терминал и для него написать редактор. Для каждого типа терминала написать программу отображения этого терминала на сетевой виртуальный терминал. Все программное обеспечение для виртуального сетевого терминала расположено на уровне приложений.

Другой пример - передача файлов. Разные операционные системы используют разные механизмы именования, представления текстовых строк и т.д. Для передачи файлов между разными системами надо преодолевать все такие различия. Для этого есть приложение FTP, также расположенное на уровне приложений. На этом же уровне находятся: электронная почта, удаленная загрузка программ, удаленный просмотр информации и т.д.

19. Организация сетевого взаимодействия. Семейство протоколов TCP/IP. Сравнение с эталонной моделью OSI ISO. Основные функции протоколов IP и TCP. Основные прикладные протоколы архитектуры TCP/IP.

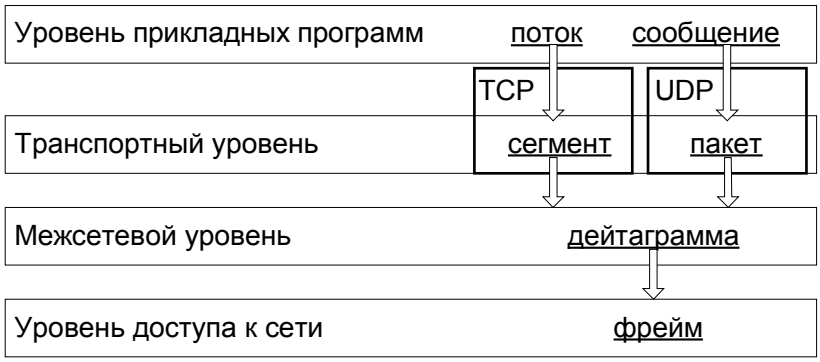
Протоколы семейства TCP/IP не следуют строго модели ISO/OSI. Они разбиты на *четыре* уровня.

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

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

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

Протоколы каждого из уровней оперируют порциями данных, имеющих зависящие от конкретного уровня названия и структуру.
Протоколы уровня доступа к сети используют при передаче и приеме данных пакеты, называемые ***фреймами.*** На межсетевом уровне используются ***дейтаграммы***. Уровень транспортных протоколов семейства представляется двумя протоколами TCP и UDP. Протокол TCP оперирует ***сегментами***. UDP – ***пакетами***. На уровне прикладных программ, системы построенные на использовании протокола TCP используют ***поток*** данных, а системы использующие UDP - ***сообщения***.



Эталонная модель TCP/IP

Прототипом для модели TCP/IP послужил прародитель всех компьютерных сетей - сеть ARPA. Эта сеть образовалась в результате НИР, проведенного по инициативе Министерства Обороны США. Позднее к этому проекту подключились сотни университетов и гос. учреждений Америки. С самого начала эта сеть задумывалась как объединение нескольких разных

сетей. Одной из основных целей этого проекта было разработать унифицированные способы соединения сетей. С появлением спутниковых и радио цифровых каналов связи проблема становилась только актуальнее. Так появилась модель TCP/IP. Свое название она получила по именам двух основных протоколов: TCP - протокол управления передачей (Transmission Control Protocol), и IP - межсетевой протокол (Internet Protocol).

Другой целью проекта ARPA было создание протоколов, независящих от характеристик конкретных хост-машин, маршрутизаторов, шлюзов и т.п.

Кроме этого связь должна поддерживаться даже если отдельные сети компоненты будут выходить из строя во время соединения. Другими словами связь должна поддерживаться до тех пор, пока источник информации и получатель информации работоспособны. Архитектура сети не должна ограничивать приложения, начиная от простой передачи файлов до предачи речи и изображения в реальном времени.

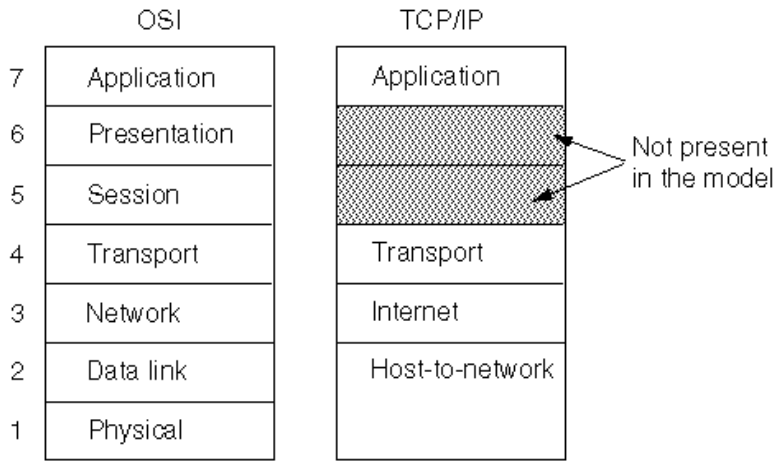


Fig. 1-18. The TCP/IP reference model.

Межсетевой уровень
В силу вышеперечисленных требований выбор очевиден: сеть с коммутацией пакетов с межсетевым уровнем без соединений. Этот уровень называется межсетевым уровнем. Он является основой всей архитектуры. Его назначение обеспечить доставку пакетов, движущихся в сети независимо друг от друга, даже если получатель принадлежит другой сети. Причем пакеты могут поступать к получателю не в том порядке, в каком они были посланы. Упорядочить их в надлежащем порядке - задача вышележащего уровня.

Межсетевой уровень определяет межсетевой протокол IP и формат пакета. Ни протокол, ни формат пакета не являются официальными международными стандартами, в отличие от протоколов эталонной модели OSI.

Итак, назначение межсетевого уровня в TCP/IP доставить IP пакет по назначению. Это как раз то, за что отвечает сетевой уровень в ISO модели.

Транспортный уровень
Над межсетевым уровнем расположен транспортный уровень. Как и OSI модели его задача обеспечить связь точка-точка между двумя равнозначными активностями. В рамках TCP/IP модели было разработано два транспортных протокола. Первый TCP: надежный протокол с соединением. Он получает поток байт, фрагментирует его на отдельные сообщения и передает их на межсетевой уровень. На машине получателе равнозначная активность TCP протокола собирает эти сообщения в поток байтов. TCP протокол также обеспечивает управление потоком.

Второй протокол UDP (User Datagram Protocol). Это ненадежный протокол без соединения для тех приложений, которые используют свои механизмы фрагментации, управления потоком. Он часто используется для передачи коротких сообщений в клиент - серверных приложениях, а также там где скорость передачи важнее ее точности.

Уровень приложений
В TCP/IP модели нет уровней сессии и представления. Необходимость в них была не очевидна для ее создателей. На сегодня дело обстоит так, что разработчик сложного приложения берет на себя проблемы этих уровней.

Над транспортным протоколом располагается уровень приложений. Этот уровень включает виртуальный терминал - TELNET, передачу файлов - FTP, электронную почту - SMTP. Позднее к ним добавились: служба имен домена - DNS (Domain Name Service) отображающая логические имена хост-машин на их сетевые адреса, протокол для передачи новостей - NNTP, и протокол для работы с гипертекстовыми документами во всемирной паутине (WWW) - HTTP.

Под межсетевым уровнем в TCP/IP модели великая пустота. Модель ничего не говорит что происходит так, лишь что хост-машина должна быть связана с сетью через некоторый протокол. Никаких ограничений на этот протокол, равно как и рекомендаций нет.

Сравнение моделей MOC и TCP/IP

Обе модели имеют много общего. Обе имеют уровневую иерархию, поддерживают понятие стека протоколов. Назначение их уровней примерно одинаково. Все уровни от транспортного и ниже используют протоколы для поддержки взаимодействия типа точка-точка, не зависящего от организации сети. Все уровни выше транспортного ориентированы на приложения.

В модели OSI центральными являются три понятия:

1) сервис
2) интерфейс
3) протокол.

Здесь можно провести аналогию с объектно-ориентированным программированием. У каждого объекта есть набор методов - сервис, которые определяют те операции, которые этот объект может выполнять. Иными словами, сервис - это семантика методов. Каждый метод имеет интерфейс - набор параметров, имя и т.п. Реализация методов скрыта в объекте - протокол; и не видима пользователю.

В TCP/IP модели нет столь же четкого выделения этих понятий. Там понятие протокола не столь четко "упрятано" и независимо от остальных частей модели. Этот факт есть следствие того как создавались эти модели. TCP/IP модель создавалась post factum, а OSI до того как появились протоколы. Поэтому понятие протокола там абсолютно не зависит от остальных частей модели. Например, изначально протоколы канального уровня в OSI создавались для соединений точка-точка. Позднее, когда появились средства типа вещания, на этот уровень были добавлены соответствующие протоколы. Никаких других изменений не последовало.

TCP/IP модель была создана когда TCP/IP стек уже существовал. Поэтому эта модель прекрасно описывала этот стек, но только этот стек и никакой другой.

Модели имеют разное число уровней. Обе имеют уровень приложений, транспортный уровень и сетевой уровень. Все остальные уровни разные.

OSI модель поддерживает на сетевом уровне как сервис с соединением, так и без соединения. На транспортном уровне этой модели поддерживается сервис только с соединением. В TCP/IP наоборот: сетевой уровень обеспечивает сервис без соединения, но транспортный - как с соединением, так и без.

Основные прикладные протоколы архитектуры TCP/IP

Межсетевой протокол - IP

Протокол IP обеспечивает негарантированную доставку пакетов Транспортного уровня (называемых транспортными протокольными блоками данных, TPDU - Transport Protocol Data Units) в пределах интерсети в режиме без установления соединения (в дейтаграммном режиме). В протоколе IP предусмотрена операция фрагментации TPDU на более мелкие пакеты в том случае, когда это необходимо, и соответственно - обратная операция сборки, выполняемые обычно в маршрутизаторах или же в целевой ЭВМ. Каждый TPDU или фрагмент снабжается IP заголовком и передается как кадр низкоуровневыми протоколами.

Заголовок IP пакетов состоит из нескольких полей, их определение приведено ниже.

1) Версия (Version) - некоторый номер, отражающий определенный этап развития протокола. Оконечные системы и маршрутизаторы должны иметь согласованые номера версий, что гарантирует корректную обработку заголовка.
2) Длина IP заголовка (IP Header Length-IHL) - длина заголовка дейтаграммы в 32 битовых словах.
3) Тип услуги (Type of Service) С помощью данного 8-ми битового поля высокоуровневые протоколы имеют возможность указать протоколу IP (точнее, соответствующему IP-обьекту) каким образом должна быть обработана конкретная дейтаграмма.
4) Длина (length) - размер в байтах всего IP-пакета, включая область данных и заголовок пакета.
5) Идентификатор (Identification) - некоторое число, которое идентифицирует данную дейтаграмму. Совместно с адресом источника содержимое поля уникально идентифицирует дейтаграмму. Фрагменты, имеющие одинаковые адрес источника и идентификатор, обьединяются вместе с учетом значения параметра "индекс фрагмента". Операция "сборки" выполняется в маршрутизаторах или в оконечных узлах.
6) Флаги (Flags) - Два младших бита трехбитового поля применяются в процессе фрагментации. Первый бит DF (Don't Fragment) указывает, является ли дейтаграмма фрагментом, второй бит MF (More Fragment) указывает, является ли данный фрагмент последним.

7) Индекс (смещение) фрагмента (Fragment Offset) - индекс фрагмента в дейтаграмме.
8) Время жизни (TTL - Time-to-Live) - счетчик, ограничивающий время существования пакета в сети. Каждый раз при прохождении транзитного маршрутизатора содержимое TTL уменьшается на 1. Когда значение TTL принимает значение 0, пакет уничтожается. Этот механизм позволяет решать проблему бесконечно зацикленных пакетов.
9) Протокол (Protocol) - транспортный протокол (например, TCP), для которого предназначается данная дейтаграмма. Большинство транспортных протоколов зарегистрированы под определенными номерами в соответствующих центрах интерсети.
10) Контрольная сумма (КС) заголовка пакета (Header Checksum) - применяется для обеспечения контроля целостности IP- заголовка.
11) Адрес Источника и Адрес Назначения (Source and Destination Addresses) - с помощью этих полей, содержащих IP-адреса, идентифицируются источник и получатель пакета. IP-адрес специфицирует положение ЭВМ в терминах сеть-машина. IP-адрес - это 32-х битовое число, для удобства чтения представляемое в десятично-точечной нотации: четыре десятичных числа, разделенных точками (например, 192.32.45.1).
12) Опции. Поле опций имеет переменный размер, что позволяет связать с IP-дейтаграммой различные необязательные услуги.

В настоящее время используются три класса IP-сетей:

1) Класс A. IP-адрес в первом байте специфицирует сеть (первый бит характеризует класс сети). Следующие 3 байта (24 бита) задают адрес ЭВМ (host) в данной сети. Этот класс сетей применим в случае сетей с большим количеством хостов. Пример - ARPANET;
2) Класс B. IP - адрес в первых двух байтах специфицирует сеть (два первых бита характеризуют класс сети). Следующие 2 байта (16 бит) задают адрес ЭВМ. Данный класс сетей применяется в случае, когда отдельные сети в интерсети обьединяют среднее число ЭВМ (университеты, коммерческие организвции);
3) Класс C. IP - адрес в первых трех байтах специфицирует сеть (три первых бита характеризуют класс сети). Последний байт (8 бит) определяет ЭВМ в сети. Этот класс является полезным в случае, когда существует небольшое число ЭВМ, образующих логически связанное объединение.

Для всех случаев начальные биты адреса сети идентифицируют класс сети. Данная схема адресации обеспечивает достаточную гибкость при описании любых типов сетей. При разработке интерсети назначаются (распределяются) только номера конкретных сетей. Причем номера сетей выбираются среди свободных на основе определенных характеристик (например, число машин, объединяемых сетью).

Протокол управления передачей данных - TCP

Протокол управления передачей данных TCP (Transmission Control Protocol) является основным транспортным пртоколом интерсети. TCP обеспечивает прием сообщений любого размера от высокоуровневых протоколов (ULP - Upper-Layer Protocols) и их полнодуплексную, подтверждаемую передачу, ориентированную на соединение с управлением потоком данных. Указанным образом передача сообщений выполняется между двумя обьектами, локализованными в двух различных станциях, подключенных к интерсети, и реализующих протокол TCP.

В соответствии с протоколом TCP данные передаются в виде неструктурированного байтового потока. Каждый байт идентифицируется последовательным номером. В целях экономии времени и оптимального использования полосы пропускания TCP оддерживает определенное число одновременных ULP-взаимодействий.

Формат пакета:

1) Порт источника (Source Port) - С помощью 16-ти битового поля Порт источника осуществляется идентификация ULP-источника.
2) Порт назначения (Destination Port)
3) Номер последовательности (Sequence Number) - тридцатидвухбитовое поле Номер последовательности обычно содержит номер первого байта данных текущего сообщения. В случае, когда сообщение разделяется на несколько частей, TCP использует Поле последовательности для корректной сборки сообщения и гарантированной его доставки высокоуровневому протоколу (ULP).
4) Подтверждаемый номер (Acknowlegment Number) - В случае, когда установлен бит ACK (определен ниже), в 32-х битовом поле Подтверждаемый номер содержится номер следующего байта данных, прием которого ожидается на стороне передатчика данного сообщения. Механизм подтверждения TCP разработан с целью наиболее эффективного использования полосы пропускания сети. Вместо того, чтобы подтверждать прием каждой порции данных, в TCP подтверждение задерживается до тех пор, пока не будет отработана целая серия актов передач, которые затем совокупно и подтверждаются.
5) Смещение данных (Data Offset) длина заголовка TCP в 32-х битовых словах. Длина заголовка является переменной, поскольку размер поля "Опции" (определено ниже) переменный.
6) Резервное (Reserved)
7) Флаги - используются для передачи управляющей информации.
8) Окно (Window) - определяет число байтов данных, начиная с номера, указанного в поле "Подтверждаемый номер", которые хотел бы принять передатчик. Данное поле, совместно с полями Номер последовательности и Подтверждаемый номер, используется при реализации механизма управления потоком данных, основанного на понятии окна.

9) Контрольная сумма (Checksum)
10) Указатель срочных данных (Urgent pointer) - смещение относительно значения поля
11) Номер последовательности, указывающее положение срочных данных. Срочные данные (urgent data) представляют собой данные, которые с точки зрения ULP считаются очень важными. Зачастую, это управляющая информация, например, сигналы прерывания с клавиатуры. В рамках TCP не предпринимается никаких действий в отношении этих данных.
12) Опции (Options). Наиболее распространенная опция - это "максимальный размер сегмента", она используется в ходе фазы установления соединения для того, чтобы определить наибольший по размеру сегмент данных, который TCP может принять (от протокола более высокого уровня - ULP).

Протокол маршрутизации (Routing information protocol - RIP)
Протокол RIP обеспечивает работоспособность протокола IP. С его помощью формируется согласованная информация о сетевых маршрутах и связности интерсетей, которая используется протокольными IP-объектами, резидированными в сетевых ЭВМ. В соответствии с протоколом RIP периодически выполняется передача текущей маршрутной информации. Маршрутная информация представляет собой список сетей назначения с указанием удаления, на котором от них находится источник данной информации. Удаление задается числом переходов (hops) по транзитным сетям до целевой сети (точнее числом промежуточных маршрутизаторов).

Межсетевой протокол управления (ICMP - Internet Control Message Protocol)
Протокол ICMP сопровождает и обеспечивает работоспособность протокола IP в части контроля за ошибками сети и ее диагностики. Это связано с тем, что протокол IP является дейтаграммным и не обеспечивает исполнение указанных функций. Протокол ICMP в этом смысле дополняет протокол IP, предоставляя протоколу TCP или другим высокоуровневым протоколам, т.е. ULP, диагностическую информацию.

Протокол передачи дейтаграмм (UDP - User Datagram Protocol)
Протокол UDP подобно протоколу TCP обеспечивает транспортный сервис. Однако в отличие от TCP в протоколе UDP отсутствует фаза установления транспортного соединения и не осуществляется подтверждение приема данных. Протокол UDP выполняет только транспортировку данных (дейтаграмм), полученных от высокоуровневых протоколов (ULP). Протокол UDP не обременен накладными расходами на установление и завершение транспортного соединения, на управление потоком данных и на обеспечение других функций TCP. В результате протокол UDP является более скоростным, чем протокол TCP. По этой же причине и, исходя из простоты реализации, протокол UDP применяется в качестве средства транспортировки данных многими ULP.

Протокол передачи файлов (File Transfer Protocol - FTP)
Протокол FTP является протоколом сетевых процессов и предоставляет пользователям возможность пересылать копии файлов между двумя ЭВМ интерсети. Протокол FTP обеспечивает также функции регистрации, проверки калалогов, исполнения команд, манипуляции с файлами и другие функции управления сеансом. Все эти функции разрабатывались таким образом, чтобы их исполнение не зависело от операционных систем ЭВМ и различий в аппаратных платформах.

20. Средства межсетевого взаимодействия (мосты, маршрутизаторы, шлюзы).

Определение. Устройства, с помощью которых формируются интерсети, называются межсетевыми устройствами.
Определение. Сегмент - это сеть ЭВМ, в которой отсутствуют межсетевые устройства.

Межсетевые устройства разделяются на группы, соответствующие уровням Модели OSI, функции которых они выполняют. Повторители работают на Физическом уровне. Мосты - на Канальном уровне. Маршрутизаторы - на Сетевом уровне. Шлюзы реализуют функции более высоких уровней Модели OSI.

Повторители
Цифровые и аналоговые сигналы, переносящие информацию, из-за неизбежного ослабления и затухания, а также из-за воздействия помех, приводящих к потере целостности данных, могут передаваться на довольно ограниченные расстояния. Задача заключается в увеличении этого расстояния. Простое усиление сигналов не является лучшим вариантом, поскольку наряду с усилением полезного сигнала усиливаются и шумы.

Повторитель выполняет эту функцию Физического уровня, репродуцируя сигналы и осуществляя их дальнейшую прозрачную передачу. Повторитель не вносит каких-либо изменений в передаваемые данные, не анализирует адресную информацию и структуру данных, относящихся к другим уровням. Только восстанавливает кондиционность принятых данных и их последующую передачу.

Повторитель выделяет и сохраняет принятые цифровые данные. Затем он реконструирует выходной сигнал и осуществляет его передачу. Новый сигнал, с большой точностью повторяющий исходный сигнал, ранее переданный источником данных, передается в следующий сегмент сети. Теоретически эта функция может повторяться необходимое число раз, однако на практике в реальных сетях число повторителей между передающей и принимающей станциями ограничено.

Мосты

Мосты используются для обьединения двух сетей на Канальном уровне. В качестве устройства Канального уровня мост имеет доступ к адресной информации Физического уровня. Другими словами, он может распознавать физические адреса источника и приемника информации, участвующих в передаче. В результате анализа физических адресов (и другой информации Канального уровня) мост или направляет данные во второй сегмент, или игнорирует их. В отличие от повторителя мост селектирует поток проходящих через него данных.
Поскольку мосты фильтруют потоки данных на основе физических адресов станций, они обычно используются для выделения перегруженных участков сети в отдельные сегменты. Такое разбиение на сегменты предохраняет внутренний трафик каждого сегмента от влияния внешних трафиков других сегментов. Поскольку межсегментный трафик не слишком велик, данная стратегия довольно эффективно снижает трафик каждого сегмента.

Мост выполняет следующие действия:

1) Принимает все данные сегмента А.
2) Игнорирует все пакеты, адресованные узлам сегмента А.
3) Переповторяет все другие пакеты во внешние сегменты через соответствующие порты.
4) Выполняет те же действия для всех других подключенных сегментов.

Существует два вида мостов: прозрачные, называемые также обучающимися (learning bridges), и программируемые или мосты с маршрутизацией потока (source-routing). Программируемые мосты применяются в сетях фирмы IBM. Во всех других сетях используются прозрачные мосты.

Прозрачные мосты не требуют какого-либо предварительного программирования. После включения они "изучают" обстановку, отображая физические адреса устройств в номера линий, к которым эти устройства подключены. Используя найденные отображения, мосты формируют таблицы "физический адрес устройства/сетевой сегмент". Эти таблицы используются мостом для выбора с помощью физического адреса назначения пакета соответствующего сегмента, в который следует направить пакет. Если найденный сегмент является сегментом, из которого был получен пакет, то мост игнорирует его. В противном случае пакет передается в соответствующую линию (сегмент).

В маршрутных сетях путь следования пакетов к определенной станции назначения указывается в заголовке передаваемых пакетов. Программируемые мосты поэтому проще, их задача сводится к передаче пакета в следующий узел, указанный в маршруте следования пакета. Этим узлом может быть другой мост или целевое устройство. В маршрутных сетях минимизируется стоимость моста, поскольку ему не нужно "обучаться". С другой стороны, все другие устройства должны располагать маршрутной информацией, описывающей доступ ко всем возможным устройствам.

Устройства в маршрутных сетях определяют пути доступа к целям, выполняя рассылку специальных пакетов, которые называются "пакеты discovery". Если устройство не получает ответа от устройства назначения на свой пакет discovery в пределах своего сетевого сегмента, то осуществляется передача пакета discovery во все другие сегменты, образующие интерсеть. В пакете discovery, проходящем сегменты интерсети, собирается маршрутная информация, описывающая путь до устройства назначения. Если существует множество путей, то по каждому пути пройдет свой пакет discovery. Устройство назначения отвечает на все принимаемые им пакеты discovery. В результате устройство-инициатор пакета discovery будет располагать маршрутной информацией, позволяющей сделать выбор лучшего пути.

Маршрутизаторы

Маршрутизаторы имеют доступ к управляющей информации трех нижних уровней Модели OSI (Физического, Канального и Сетевого). Информация третьего уровня обычно описывает, так называемые, логические сетевые адреса. Логические адреса, в отличие от физических адресов, обычно назначаются элементам сети соответствующими административными службами. Именно это отличает первые от вторых.

Физические адреса обычно назначаются производителями оборудования и жестко связаны с соответствующей аппаратурой, в отличие от логических адресов. Физические адреса, как правило, уникальны. С другой стороны, логические адреса могут использоваться сетевой администрацией в целях образования групп устройств, обладающих схожими свойствами (например, принадлежность одному отделению фирмы в рамках здания). Логические адреса по сравнению с физической обеспечивает большую гибкость, поскольку могут быть легко изменены и организованы в иерархические группы.

Маршрутизаторы передают информацию в интерсеть, используя логические адреса. Логически сеть разделяется на так называемые подсети. Подсети могут охватывать один и более сетевых сегментов.

Маршрутизаторы отличаются от мостов тем, что они используют для своей работы логические, а не физические адреса. Маршрутизаторы также применяют один (или более) специальных алгоритмов маршрутизации для вычисления лучшего пути в интерсети. Пути могут вычисляться динамически (в реальном масштабе времени) так, чтобы в большей степени соответствовать изменяющимся условиям в сети.

Алгоритмы динамической маршрутизации отличаются набором тех факторов (метриками), которые принимаются к рассмотрению при вычислении лучшего пути. Например, в одном алгоритме лучший путь определяется на основе минимизации числа переходов до устройства назначения (минимальная длина пути). В другом алгоритме в качестве метрики может использоваться время передачи. Алгоритмы маршрутизации через модемы рассматривают целое множество различных взвешенных факторов. В ряде случаев сетевые администраторы могут изменять веса элементов метрики

(например, стоимость использования линии) для того, чтобы настраивать систему в соответствии с изменениями, носящими внешний характер.

Маршрутизаторы выполняют гораздо более интенсивную вычислительную работу, чем мосты. Поэтому их производительность, измеряемая обычно числом переданных пакетов в секунду, не слишком велика. Однако, с другой стороны, маршрутизаторы способны осуществлять довольно сложные вычисления для выбора пути в соответствии с алгоритмами маршрутизации. Существуют и другие их положительные свойства. В связи с этим решение покупать мост или маршрутизатор зависит от конкретных потребностей администрации сети и от существующего сетевого окружения.

Многие модемные маршрутизаторы в действительности обеспечивают и функции мостов (такие маршрутизаторы называют М - маршрутизаторами). Большинство алгоритмов маршрутизации являются специальными протоколами сетевого уровня. М-маршрутизаторы поддерживают наиболее известные протоколы (и их алгоритмы выбора пути), но, кроме этого, могут работать с пакетами, для которых такого рода поддержка не обеспечивается. Другими словами, М-маршрутизатор проверяет принятый пакет, пытаясь определить, поддерживается ли он каким-либо из имеющихся алгоритмов маршрутизации. Если пакет не поддерживается ни одним из алгоритмов маршрутизации, то он обрабатывается в соответствии с правилами моста с использованием содержащейся в нем информации Канального уровня.

Некоторые производители мостов пытаются достичь компромисса с другой стороны, реализуя в своих изделиях некоторые функции маршрутизаторов. Такие устройства обычно называют маршрутными мостами (routing bridges). Маршрутные мосты могут выполнять некоторые минимальные функции выбора пути и обеспечить достаточную безопасность маршрутов передачи данных. И все же из-за того, что эти устройства не имеют доступа к информации 3-его уровня Модели OSI, они не могут также успешно выполнять процедуры выбора пути, как это делают маршрутизаторы.

Шлюзы

Ни одно из устройств, которые мы рассмотрели выше, не решает проблемы объединения двух и более подсетей, которые имеют различные протоколы над Сетевым уровнем. Поскольку малые разнородные сети интегрируются в большие гетерогенные сети, для обеспечения их работоспособности необходимо решить проблему взаимодействия различных протокольных уровней. Устройство, для связи подсетей должно осуществлять трансляцию (преобразование) одних протокольных соглашений в другие. Преобразование протокольных соглашений является функцией шлюза.

Шлюзы изготавливаются или в виде отдельного конструктива, или в виде программно-аппаратных компонентов, которые встраиваются в существующие ЭВМ. Устройства, выполненные в виде отдельного конструктива, более дорогие, но в то же время обладают большей производительностью. Устройства со встроенными компонентами могут функционировать или в выделенном режиме (т.е. в режиме исключительного исполнения функций шлюза), или в совмещенном режиме (функции шлюза исполняются наряду с решением другого рода задач).

21. Методы защиты от несанкционированного доступа в компьютерных сетях.

7.1. Сетевая безопасность

Пока сети использовались лишь в университетах для научных исследований, а в крупных организациях для совместного использования устройств, например, принтеров, вопросы безопасности сетей просто не возникали. Теперь, кода сетью пользуется обыватель для управления банковским счетом, покупки, уплаты налогов – безопасность становится серьезной проблемой.

Проблема безопасности сети очень многогранна и охватывает широкий спектр вопросов. Большую их часть можно разделить на следующие группы:

4. Секретность
   - несанкционированный доступ к информации (никто не может прочесть ваши письма);
   - несанкционированное изменение информации (никто без вашего разрешения не может изменить данные о вашем банковском счете);
2. Идентификация подлинности пользователей
   - имея с кем-то дело через сеть, вы должны быть уверены, что это тот, за кого он себя выдает (если вы получили сообщение от налоговой инспекции уплатить определенную сумму денег, вы должны быть уверены, что это не шутка).
3. Идентификация подлинности документа
   - как правило, это проблема подписи;
4. Надежность управления
   - несанкционированное использование ресурсов (если вы получите счет за телефонные переговоры, которые вы не делали, вам это вряд ли понравится);

Разные люди по разным мотивам пытаются нарушить безопасность сети. На **рис.7-1** показан список категорий людей и возможная их мотивация.

Adversary	Goal
Student	To have fun snooping on people's email
Hacker	To test out someone's security system; steal data
Sales rep	To claim to represent all of Europe, not just Andorra
Businessman	To discover a competitor's strategic marketing plan
Ex-employee	To get revenge for being fired
Accountant	To embezzle money from a company
Stockbroker	To deny a promise made to a customer by email
Con man	To steal credit card numbers for sale
Spy	To learn an enemy's military strength
Terrorist	To steal germ warfare secrets

Fig. 7-1. Some people who cause security problems and why.

Рис. 7-1.

Глядя на эти четыре группы проблем, не трудно видеть что и в обычных системах людям приходится иметь дело с таким проблемами. Распознание подделки документов, ограничение доступа к информации (уровни секретности), опознание людей (биометрические методы) и т.д.

Прежде чем приступить к рассмотрению методов решений перечисленных проблем рассмотрим где, в каком месте стека протоколов должно располагаться обеспечение безопасности, защита сети. Одного такого места нет. Каждый уровень способен сделать свой вклад. Например, на физическом уровне, чтобы контролировать доступ к физическому каналу, можно поместить кабель в опечатанную трубу, заполненную газом под давлением. Любая попытка просверлить трубу приведет к падению давления газа и срабатыванию датчика давления. Это, в свою очередь, включит сигнал тревоги.

На канальном уровне данные могут быть зашифрованы на одной машине и расшифрованы на другой. Об этом шифре верхние уровни могут ничего не знать. Однако, поскольку пакет дешифруется на каждом маршрутизаторе, то там он может стать предметом атаки. Тем не менее, этот метод, называемый шифрованием канала, часто применяется в сетях.

На сетевом уровне распространенным решением является застава (firewall). На транспортном уровне проблема секретности хорошо решается шифрованием всех сегментов транспортного соединения. Однако, до сих пор нет удовлетворительного решения для проблемы идентификации пользователя и идентификации документа.

7.1.1. Обычное шифрование

История шифрования богата и разнообразна. Традиционно ее развивали четыре группы людей – военные, дипломаты, любители вести дневники и любовники.

Стандартная схема шифрования такова (**рис.7-2**). Исходный текст, называемый также открытым текстом (plain text), обрабатывается специальной функцией, со специальным параметром, называемым ключом. Получается, так называемый, шифртекст (ciphertext) или криптограмма. Злоумышленник аккуратно копирует все шифртексты. Однако, в отличие от получателя у него нет ключа и он не может быстро прочесть сообщение. Иногда, злоумышленник может не только просто копировать сообщение, но позже отправлять свои, имитируя настоящего отправителя, чьи сообщения он копировал. Такого злоумышленника называю активным. Искусство создания шифра называют криптографией, а вскрытия – криптоанализом. Обе эти дисциплины образуют криптологию.

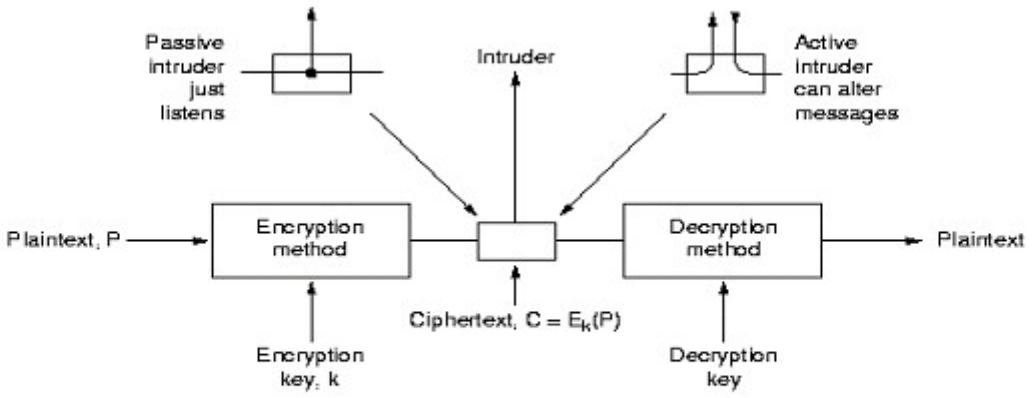


Fig. 7-2. The encryption model.

Рис. 7-2.

Основное правило шифрования - криптоаналитик знает основные приемы шифрования. Смена метода шифрования, его создание, тестирование, внедрение – всегда сопровождаются огромными затратами. Как часто надо менять шифр? Как определить, что шифр уже вскрыт?

Одно из решений – шифрование на основе ключей. Ключ – относительно короткая строка текста, которая используется при шифровании и расшифровке сообщений. Тогда, вся схема шифрования всем хорошо известна, менять ничего не надо, надо лишь время от времени изменять ключи.

Публикуя алгоритм шифрования, его автор получает задаром консультации многих исследователей в этой области. Если ни один из них в течении 5 лет не объявил, что он вскрыл алгоритм, то такой алгоритм можно считать вполне надежным.

Основа секретности – ключ. Его длина – один из основных вопросов разработки. Рассмотрим, как пример, комбинационный цифровой замок. Все знают, что его ключ – последовательность цифр. Для замка из двух цифр, надо перебрать 100 комбинаций, для 3 – 1000 и т.д. Длина ключа – объем работы, который надо проделать криптоаналитику, чтобы вскрыть шифр. Этот объем растет экспоненциально от длины ключа. Секретность достигается открытостью алгоритма и длиной ключа. Чтобы защититься от чтения почты 64 разрядного ключа вполне достаточно. Для секретности государственных документов потребуется 128 или даже 256 разрядов.

С точки зрения криптоаналитика проблема дешифровки возникает в трех вариантах:

- есть только шифрограмма;
- есть шифрограмма и само сообщение;
- есть фрагменты исходного сообщения и их шифрограммы.

Шифрование замещением

Все приемы шифрования исторически делились на шифрование замещением и шифрование перестановкой.

Шифрование замещением состоит в том, что буква или группа букв замещается другой буквой или группой букв. Например, шифр Юлия Цезаря состоял в замене каждой буквы третьей следующей за ней в алфавите.

а б в г д е ё ж з и й к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я
г д е ё ж з и й к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я а б в

пришёл увидел победил
тулыио целжзо тсдзжло

Это, так называемое, моноалфавитное замещение, где ключом является 33 буквенная строка, соответствующая алфавиту. Здесь возможно 26! = 4×10^{26} ключей. Даже если на применение одного ключа компьютер будет тратить 1 μ сек, то на расшифровку уйдет около 10^{13} лет. Однако если применить знания частотных характеристик языка, а именно частоту встречаемости отдельных букв, двух буквенных буквосочетаний, трехбуквенных сочетаний и т.д., то решение можно получить быстрее. Надо подсчитать частоту букв в шифртексте и попытаться сопоставить наиболее часто встречающимся буквам в шифре сопоставить наиболее часто встречающиеся в языке. Затем найти устойчивые буквосочетания и т.д. Поэтому важное значение имеют дополнительные сведения: на каком языке написано исходное сообщение, его длина. Чем длиннее сообщение, тем представительнее выборка для анализа по встречаемости букв, буквосочетаний.

Шифрование перестановкой

Шифрование перестановкой состоит в изменении порядка букв без изменения самих букв. Один из таких методов – шифрование по столбцам. Выбираем ключ – последовательность неповторяющихся символов. Символы в этой последовательности нумеруются в соответствии с их местом в алфавите. Номер один получает

буква, расположенная ближе всего к началу алфавита, номер два, следующая за ней и т.д. Чем ближе к началу алфавита символ, тем меньше его номер. Шифруемый текст размещается по строкам. Длина строки – длина ключа. Получаем массив. Столбцы нумеруются в соответствии с ключом. Каждому столбцу соответствует символ ключа, который имеет определенный номер. Упорядочим столбцы по возрастанию этих номеров. Все символы первого столбца выписываются первыми, затем символы второго и т.д. (**рис.7-3**). Этот метод можно усовершенствовать многими способами.

Fig. 7-3. A transposition cipher.

Рис. 7-3.

Для раскрытия этого типа шифров криптоаналитик, прежде всего, должен убедиться, что он имеет дело с шифрованием перестановкой. Для этого он должен подсчитать частоту встречаемости букв в шифре. Если она соответствует частоте букв в языке, то это означает, что он имеет дело именно с перестановкой. Намек на порядок столбцов могут дать устойчивые буквосочетания в языке.

Одноразовые подложки

Построить не раскрываемый шифр достаточно просто. Выберем случайным образом битовую строку. Текст представим как битовую строку. Выполним над этими строками операцию «исключающее или» EXCLUSIVE OR. Полученная новая строка и есть криптограмма.

Этот метод, называемый методом одноразовой подложки, имеет ряд недостатков. Трудно запомнить ключ. Его где-то надо записать или носить с собой. Это делает метод уязвимым. Объем передаваемых данных по этому методу ограничен длиной ключа. Метод так же очень чувствителен к потере символов при передаче. Правда, используя возможности компьютеров, этот метод вполне применим в некоторых случаях. Набор одноразовых подложек можно записать на CD и закамуфлировать под запись последних хитов.

Рассеивание и перемешивание

Основная угроза раскрытия текста при криптоанализе состоит в высокой избыточности естественного языка. Например, частотные характеристики встречаемости букв, устойчивые буквосочетания, приветствия и т.д. В связи с этим Шеннон предложил два основных криптографических метода: рассеивание и перемешивание.

Цель рассеивания состоит в перераспределении избыточности исходного языка на весь исходный текст. Этот прием может быть реализован как перестановкой по некоторому правилу, так и замещением. Последнее, например, достигается тем, что замещающая комбинация, зависит не только от замещаемой буквы, но от ей предшествующих букв.

Цель перемешивания состоит в том, чтобы сделать зависимость между ключом и шифртекстом настолько сложной, на сколько это возможно. Криптоаналитик на основе анализа шифртекста не должен получать сколь-нибудь полезной информации о ключе. Этот метод как раз реализуется с помощью перестановок.

Следует учитывать, что применение порознь ни рассеивания, ни перемешивания не дает желаемого результата. Их совместное использование делает криптосистему намного более стойкой.

7.1.2. Два основных принципа шифрования

Методов шифрования существует множество. Некоторые из них мы рассмотрим через несколько страниц. Однако, есть два важных, основополагающих принципа, которые должны соблюдаться в каждом методе. *Первый, все шифруемые сообщения должны иметь избыточность*, т.е. информацию, которая не нужна для понимания сообщения.

Эта избыточность позволит нам отличить нормально зашифрованное сообщение от подсунутого. Пример с уволенным сотрудником, который может наслать ложных сообщений, если он знает структуру служебных сообщений в компании. Избыточность позволит обнаружить подделку. Например, если в заказах на поставку после имени заказчика стоит 3 байтовое поле заказа (2 байта – код продукта и 1 байт количество), зашифрованное с помощью ключа, то злоумышленник, прихватив с работы справочник заказчиков, может

устроить компании «веселую жизнь», сгенерировав от имени заказчиков из справочника заявки, где последние 3 байта – случайные числа. Если же заявку сделать не 3, а, например, 12 байтов, где первые 9 байтов – 0 и шифровать эту избыточную запись, то уже случайными числами здесь обойтись трудно, подделку легко распознать.

Однако такая избыточность имеет недостаток, эта избыточность может служить для криптоаналитика дополнительной информацией. Например, если в первом случае догадка о ключе и применение этого ключа к записи не дает криптоаналитику дополнительной информации о правильности ключа, то во втором случае, если в результате применение ключа-догадки, мы получим запись из 9 нулей с последующими данными, то это уже будет дополнительной информацией о правильности догадки.

Второй – надо позаботиться о специальных мерах от активного злоумышленника, который может копировать, а потом пересылать модифицированные копии. Например, временная метка позволит обнаружить сообщения, которые где-то были задержаны по непонятным причинам.

7.1.3. Алгоритмы с секретными ключами

Современная криптология использует те же идеи, что и раньше. Однако акценты расставлены иначе. Если раньше алгоритм был прост, а вся сложность заключалась в ключе, то теперь наоборот стараются алгоритм делать как можно изощреннее. Его стараются делать таким, чтобы, если криптоаналитик получил как угодно много зашифрованного текста, он не смог ничего сделать.

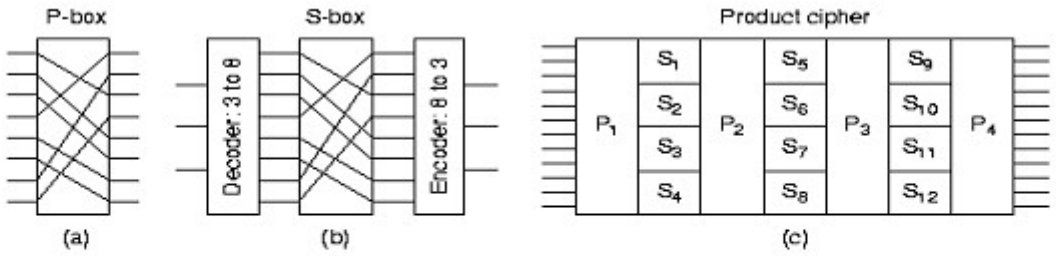


Fig. 7-4. Basic elements of product ciphers. (a) P-box. (b) S-box. (c) Product.

Рис. 7-4.

Перестановка и замещение реализуются простыми схемами, показанными на **рис.7-4**. P и S схемы могут объединять в сложные каскады. В этом случае выход становится очень сложной функцией входа. На этом рисунке P схема выполняет перестановку над словом из 8 разрядов. Схема S действует несколько сложнее, выполняя операцию замещения. Она кодирует трех разрядное слово одной из 8 линий на выходе, устанавливая ее в 1. Затем схема P переставляет эти 8 разрядов, после чего S схема выполняет замещение 8 на 3.

Алгоритм DES

В январе 1977 правительство США приняло стандарт в области шифрования (Data Encryption Standard), созданный на базе разработки фирмы IBM. На **рис.7-5** показана схема этого алгоритма.

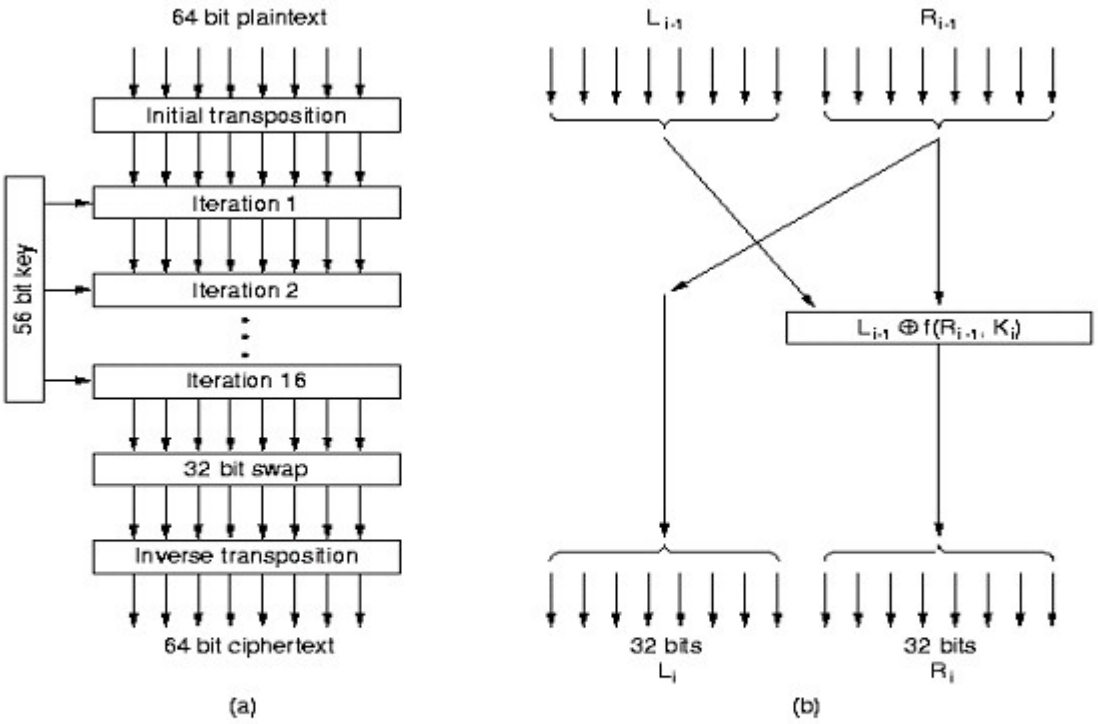


Fig. 7-5. The data encryption standard. (a) General outline. (b) Detail of one iteration.

Рис. 7-5.

Алгоритм состоит из 19 этапов. На первом этапе исходный текст разбивается на блоки по 64 бит каждый. Над каждым блоком выполняется перестановка. Последний этап является инверсией первой перестановки. Предпоследний этап состоит в обмене местами 32 самых левых битов с 32 самыми правыми битами.

Из исходного ключа с помощью специального преобразования строят 16 частных ключей для каждого промежуточного этапа алгоритма со второго по семнадцатый.

Все промежуточные этапы схожи. У них два входа по 32 бита. Правые 32 бита входа копируют на выход, как левые 32 бита. Над каждым битом из правых 32 битов выполняется преобразование, которое состоит из EXCLUSIVE OR с соответствующим левым битом и значением специальной функции над правым битом и частным ключом данного этапа. Вся сложность алгоритма заключена в этой функции.

Функция состоит из четырех шагов. На первом строят 48 разрядный номер E, как расширение 32 разрядного Ri-1 в соответствии с определенными правилами дублирования и перестановки. Над полученным номером и ключом данного шага выполняется операция EXCLUSIVE OR – это второй шаг. Результат разбивается на 8 групп по 6 бит в каждой. Каждая группа пропускается через свой S-box с четырьмя выходами каждый. На последнем шаге 8x4 бит пропускаются через P-box.

На каждой из 16 итераций используется свой ключ. До начала итераций к исходному ключу применяют 56 разрядную перестановку. Перед каждой итерацией ключ разбивают на две части по 28 разрядов каждая. Каждую часть циклически сдвигают влево на число разрядов равное номеру итерации. Ki получают как 48 разрядную выборку из 56 разрядного ключа, полученного циклическими сдвигами с дополнительной перестановкой.

У предложенного алгоритма есть два недостатка. Предложенный алгоритм – это моноалфавитное замещение с 64 разрядным символом. Всегда, когда одни и те же 64 разряда исходного текста подают на вход, одни и теже 64 разряда получают на выходе. Это свойство может использовать криптоаналитик. Одни и те же поля исходного текста попадут в одни и те же места шифра. Этим можно воспользоваться. (Пример с премиями. Один сотрудник может приписать себе премию другого, если знает взаимное расположение соответствующих полей в исходной записи. Шифр ему знать ни к чему (**рис.7-6**).)

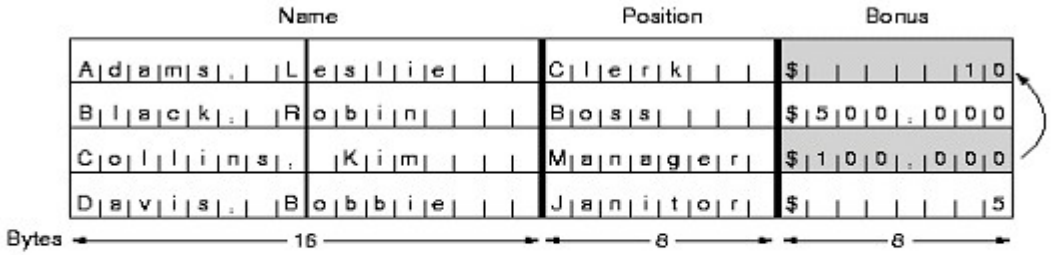


Fig. 7-6. The plaintext of a file encrypted as 16 DES blocks.

Рис. 7-6.

Второй недостаток – для начала шифрования надо иметь сразу весь 64-разрядный блок исходного текста. Это не совсем удобно, когда приходится иметь дело с интерактивными приложениями.

Первый недостаток устраняется модификацией алгоритма DES, в которой очередной блок исходного текста через операцию EXCLUSIVE OR смешивается с шифром предыдущего блока. Для первого блока используется специальный инициирующий блок, который передается вместе с шифром (**рис.7-7**).

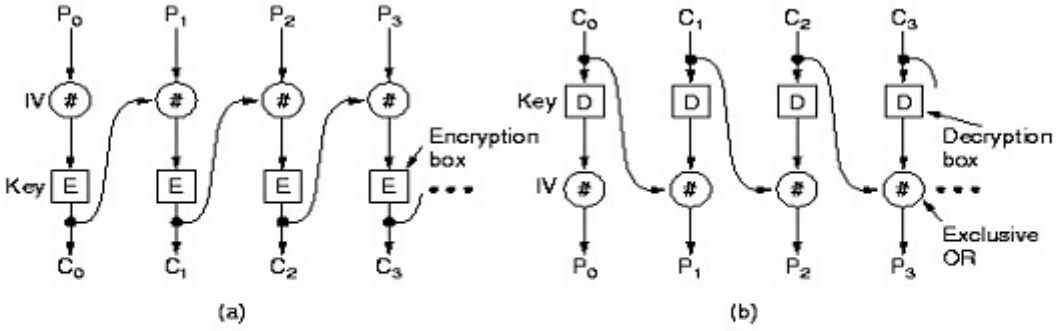


Fig. 7-7. Cipher block chaining

Рис. 7-7.

Для устранения второго недостатка используется, так называемая, обратная связь по шифру (**рис.7-8**). На **рисунке 7-8(а)** показана ситуация, когда первые 10 байт текста уже были закодированы, а последние 8 из них сохранены в сдвиговом регистре. Когда поступает очередной байт, выбирается самый левый байт 64 разрядного шифра и над ними выполняется EXCLUSIVE OR, результат посылается по линии, как шифр байта. Этот шифрованный байт посылается в сдвиговый регистр, а левый байт выталкивается из регистра. На стороне получателя над поступившим байтом выполняют ту же самую операцию. Этот алгоритм работает хорошо, пока оба сдвиговых 64 разрядных регистра одинаковы. Если при передаче шифрованного байта произойдет однобитовая ошибка, то пока испорченный байт не вытолкнут из регистра, расшифрованный текст пойдет с ошибкой. После этого ошибка исчезнет. Таким образом, одна битовая ошибка приведет к порче 64 бит текста. Этот алгоритм также предполагает наличие некоторого инициирующего блока, который обеспечивает работу алгоритма, пока все 8 правых байт не поступили.

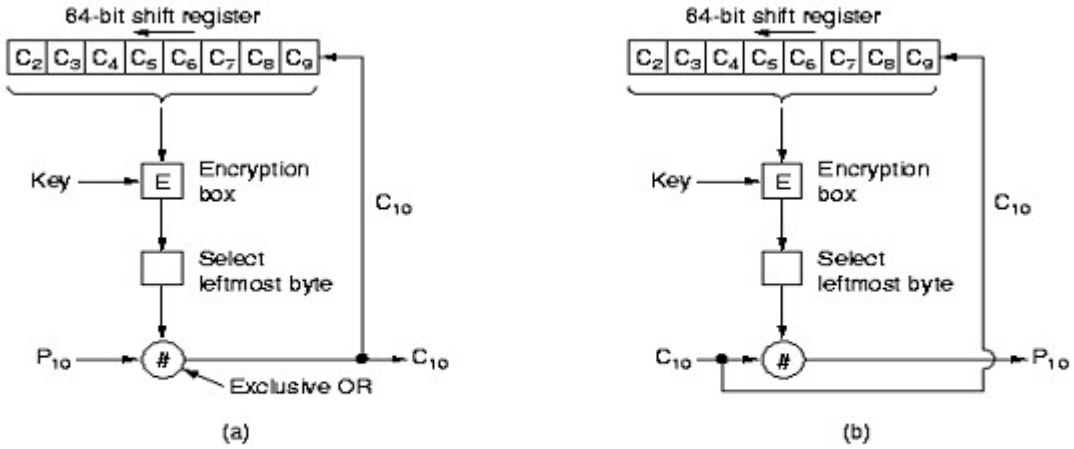


Fig. 7-8. Cipher feedback mode.

Рис. 7-8.

Раскрытие DES

Появление DES с первых же дней сопровождалось казусами. Во-первых, он был построен на базе шифра IBM, называемого Люцифер. Разница была в том, что IBM использовала 128 разрядный ключ, а не 56, как DES. Всякое появление нового шифра инициирует дискуссию с правительственным ведомством – NSA в США, у нас – ФАПСИ.

IBM засекретило, как был разработан DES. В 1977 году Диффи и Хеллман из Стэнфорда опубликовали проект машины стоимостью в 20 миллионов долларов, которая вскрывала DES. Достаточно было подать на вход этой машине небольшой фрагмент зашифрованного текста и соответствующий фрагмент исходного текста, как эта машина в течении дня находила ключ. Существует много способов атаковать шифр. Все они основаны на распараллеливании перебора множества возможных ключей. Например, 140 000 человек вооруженных компьютерами, способными выполнять 250 000 шифрований с секунду, смогут перебрать пространство 7x1016 ключей за месяц. Основной вывод - DES не является надежным шифром и его нельзя использовать для ответственных документов. Удвоение длины ключа до 112 бит кардинально меняет ситуацию. Теперь если использовать даже миллиард аппаратных дешифраторов, выполняющих по миллиарду оп.сек., потребуется 100 миллионов лет, чтобы перебрать пространство 2112=5x1033 112 разрядных ключей. Эти рассуждения могут натолкнуть на идею увеличения длины ключа, за счет двукратного применения DES с разными ключами К1 и К2.

Однако, в 1981 году Хеллман и Меркл обнаружили, что просто использование двух ключей не дает надежной схемы. Дело в том, что применение дешифрации к зашифрованному тексту дает шифр, который получается после применения первого ключа к исходному тексту. Эту мысль поясняют следующие формулы

$$Ci = EK2\ (EK1\ (Pi\ ));\ \ DK2\ (Ci\ ) = EK1\ (Pi\ )$$

В этом случае можно предложить следующую процедуру взлома:

- вычислить все возможные применения функции Е к шифруемому тексту;
- вычислить все возможные дешифрации зашифрованного текста однократным применением дешифрирующей функции;
- отсортировать полученные таблицы и искать совпадающие строки;
- полученная пара номеров строк – пара ключей;
- проверить эту пару на совпадение шифрования; если неудачный результат, продолжить с шага 1.

Однако, тройное шифрование совершенно меняет дело. На **рис.7-9** показана модификация схемы шифрования с двумя ключами в три этапа - EDE схема. Ее никому еще не удалось вскрыть. Она была положена в основу международного стандарта. Здесь может возникнуть два вопроса. Первый: почему в этой схеме используются 2, а не 3 ключа. Второй: почему используется схема EDE, а не EEE? Ответ на первый вопрос состоит в том, что двух ключей более чем достаточно для большинства применений. Использование схемы EDE вместо EEE связано с особенностями организации алгоритма DES.

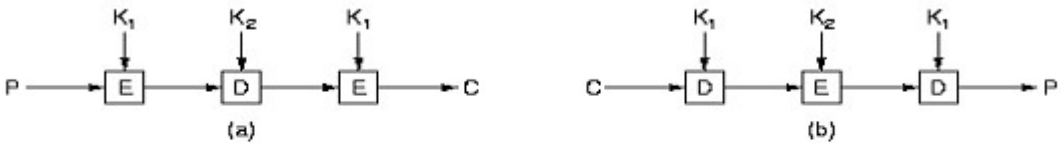


Fig. 7-9. Triple encryption using DES.

Рис. 7-9.
Надо отметить, что было предложено много других алгоритмов шифрования.

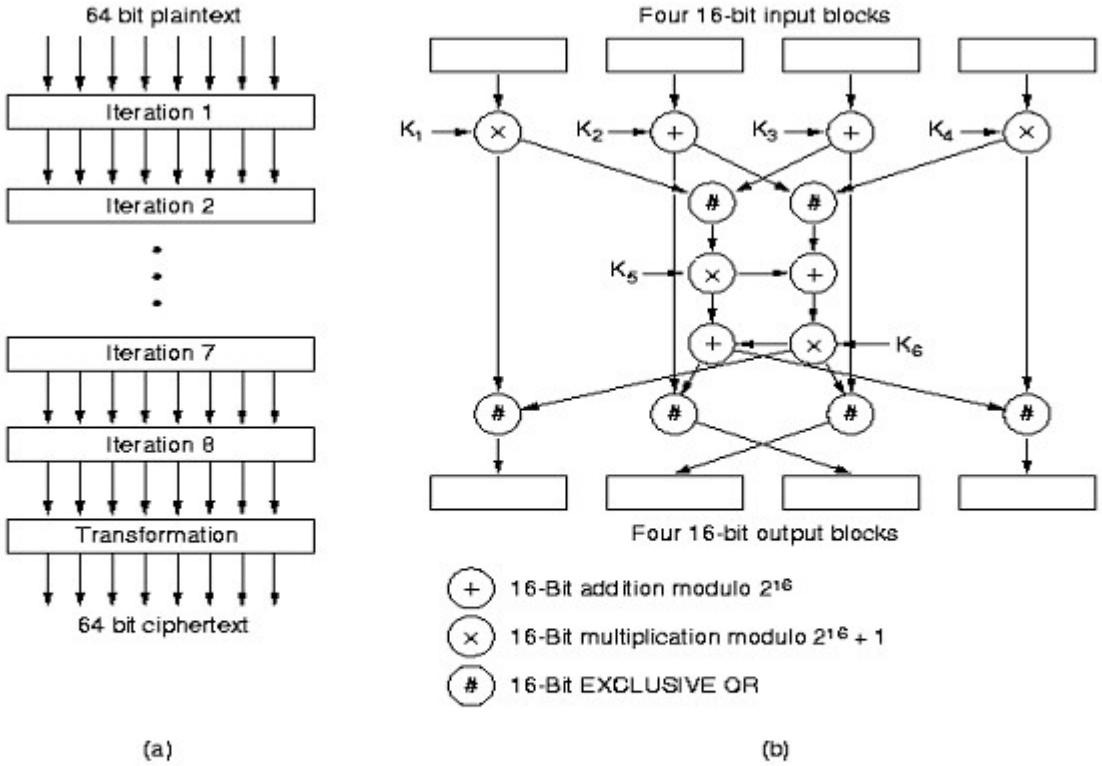


Fig. 7-10. (a) IDEA. (b) Detail of one iteration.

Рис. 7-10.

7.1.4. Алгоритмы с открытыми ключами

Идея алгоритмов шифрования с открытыми ключами была предложена в 1976 году Диффи и Хеллманом и состоит в следующем. Пусть у нас есть алгоритмы E и D, которые удовлетворяют следующим требованиям:

- D(E(P))=P;
- Чрезвычайно трудно получить D из E;
- E нельзя вскрыть через анализ исходных текстов.

Алгоритм шифрования E и его ключ публикуют или помещают так, чтобы каждый мог их получить, алгоритм D так же публикуют, чтобы подвергнуть его изучению, а вот ключи к последнему хранят в секрете. В этом случае взаимодействие двух абонентов A и B будет выглядеть следующим образом. Пусть A хочет послать B сообщение

P. A шифрует E^B (P), зная алгоритм и открытый ключ для шифрования. B, получив E^B (P), использует D^B с

секретным ключом, т.е. вычисляет D^B (E^B (P))=P. Никто не прочтет P кроме A и B, т.к. по условию алгоритм E^B не раскрываем по условию, а D^B, не выводим из E^B.

Примером такого алгоритма является алгоритм RSA, предложенный Ривестом, Шамиром и Адлеманом в 1978 году. Общая схема этого алгоритма такова

1. Выберем два больших (больше 10^{100}) простых числа p и q.
2. Вычислим n=pxq и z=(p-1)x(q-1);
3. Выберем d относительно простое к z.
4. Вычислим e такое, что exd=1 mod z.

Разбиваем исходный текст на блоки P так, чтобы каждый блок, как число не превосходил n. Для этого выбираем наибольшее k такое, чтобы $P=2^k$<n. Вычисляем $C=P^e$ (mod n), чтобы зашифровать сообщение P. Для расшифровки вычисляем $P=C^d$ (mod n). Для шифрования нам нужны (e,n) – это открытый ключ, для расшифровки (d,n) – это закрытый ключ. Можно доказать, что для любого P в указанном выше диапазоне, функция шифрования и дешифрования взаимнообратны. Безопасность этого метода основана на высокой вычислительной сложности операции разложения на множители больших чисел. Так, например, разложение на множители 200 разрядного числа потребует 4 миллиардов лет.

Plaintext (P)			Ciphertext (C)		After decryption	
Symbolic	Numeric	P^3	P^3 (mod 33)	C^7	C^7 (mod 33)	Symbolic
S	19	6859	28	13492928512	19	S
U	21	9261	21	1801088541	21	U
Z	26	17576	20	1280000000	26	Z
A	01	1	1	1	1	A
N	14	2744	5	78125	14	N
N	14	2744	5	78125	14	N
E	05	125	26	8031810176	5	E
Sender's computation			Receiver's computation			

Fig. 7-11. An example of the RSA algorithm.

Рис. 7-11.

На **рис.7-11** дан простой учебный пример применения RSA алгоритма для p=3, q= 11, n=33, z = 20, d=7 откуда e= 3.

Один из основных недостатков алгоритма RSA – он работает медленно.

7.1.5. Протоколы установления подлинности

Протоколы установления подлинности (аутентификации) позволяют процессу убедиться, что он взаимодействует с тем, кто должен быть, а не с тем, кто лишь представляется таковым.

Очень часто путают проверку прав на выполнение тех или иных операций с аутентификацией. (В первом случае имеем авторизацию.) Аутентификация отвечает на вопрос: как убедиться, что вы взаимодействуете именно с определенным процессом. Если, например, к серверу процесс обратился с запросом удалить файл x.old и объявил себя процессом Вася, то сервер должен убедиться, что перед ним действительно Вася и что Вася имеет право делать то, что просит. Ключевым конечно является первый вопрос, ответ на второй вопрос – это дело просмотра таблицы.

Общая схема всех протоколов аутентификации такова: сторона A и сторона B начинают обмениваться сообщениями между собой или с Центром раздачи ключей (ЦРК). ЦРК всегда надежный партнер. Протокол аутентификации должен быть устроен так, что даже если злоумышленник перехватит сообщения между A и B, то ни A ни B не спутают друг друга с злоумышленником. Обмен данными между A и B будет происходить по алгоритму с закрытым ключом, а вот устанавливаться соединение по алгоритму с открытым ключом.

Аутентификация на основе закрытого разделяемого ключа.

Основная идея нашего первого протокола аутентификации, так называемого *протокола ответ по вызову*, состоит в том, что одна сторона посылает некоторое число (вызов), другая сторона, получив это число, преобразует его по определенному алгоритму и отправляет обратно. Посмотрев на результат преобразования и зная исходное число, инициатор может судить правильно ли сделано преобразование или нет. Алгоритм преобразования является общим секретом взаимодействующих сторон. Будем предполагать, что стороны A и B

имеют общий секретный ключ KAB. Этот секретный ключ взаимодействующие стороны как-то установили заранее, например, по телефону. Описанная выше процедура показана на **рис.7-12**.

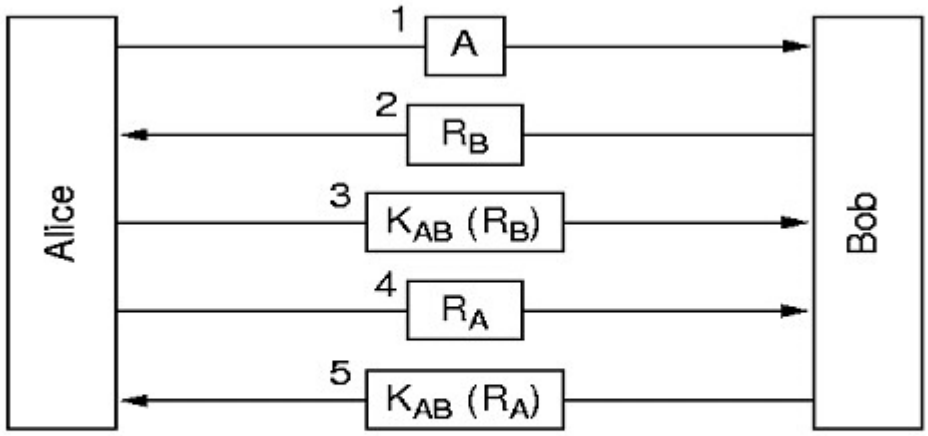


Fig. 7-12. Two-way authentication using a challenge-response protocol.

Рис. 7-12.

На этом рисунке:

A,B - идентификаторы взаимодействующих сторон;

R^i - вызов, где индекс указывает кто его послал;

K^j - ключ, индекс которого указывает на его владельца.

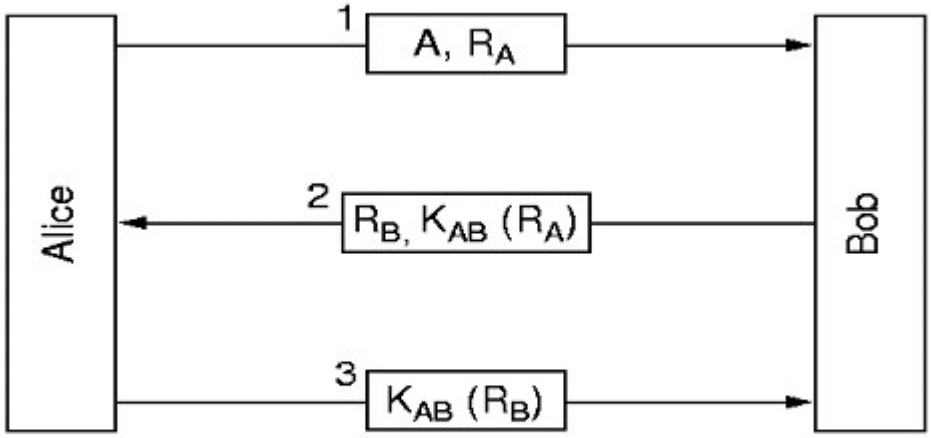


Fig. 7-13. A shortened two-way authentication protocol.

Рис. 7-13.

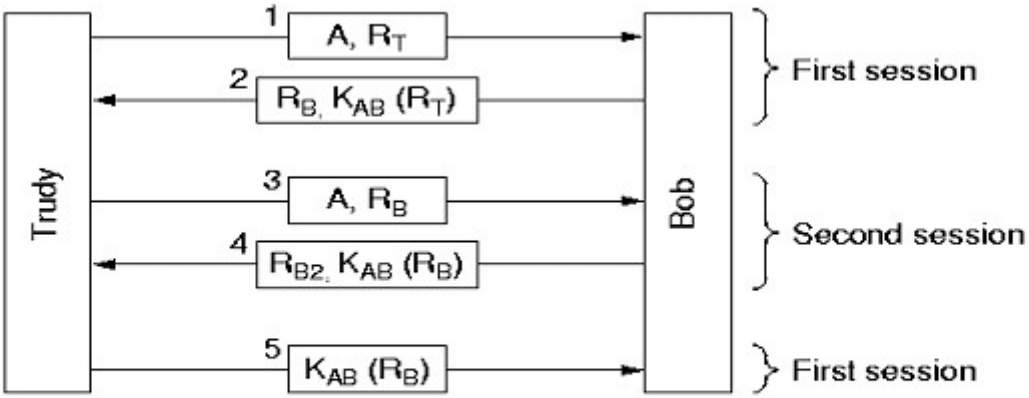


Fig. 7-14. The reflection attack.

Рис. 7-14.

На **рис.7-13** дана схема, где сокращено количество передач между сторонами, по сравнению с **рис.7-12**, а **рис.7-14** показывает «дыру» в схеме **7-13** и как злоумышленник может этой дырой воспользоваться. Это, так называемая, атака отражением. Есть несколько общих правил построения протоколов аутентификации (протокол проверки подлинности или просто подлинности):

1. Инициатор должен доказать кто он есть прежде, чем вы пошлете ему какую-то важную информацию.
2. Инициатор и отвечающий должны использовать разные ключи.
3. Инициатор и отвечающий должны использовать начальные вызовы из разных непересекающихся множеств.

В схеме на **рис.7-13** все эти три правила нарушены.

Установка разделяемого ключа.

До сих пор мы предполагали, что A и B имеют общий секретный ключ. Рассмотрим теперь, как они могут его установить? Например, они могут воспользоваться телефоном. Однако, как B убедиться, что ему звонит именно A, а не злоумышленник? Можно договориться о личной встрече, куда принести паспорт и прочее, удостоверяющее личность. Однако есть протокол, который позволяет двум незнакомым людям установить общий ключ даже при условии, что за ними следит злоумышленник.

Это протокол обмена ключом Диффи-Хеллмана. Его схема показана на **рис.7-15**. Прежде всего A и B должны договориться об использовании двух больших простых чисел n и g, удовлетворяющих определенным условиям. Эти числа могут быть общеизвестны. Затем, A выбирает большое число, скажем x, и хранит его в секрете. То же самое делает B. Его число – y.

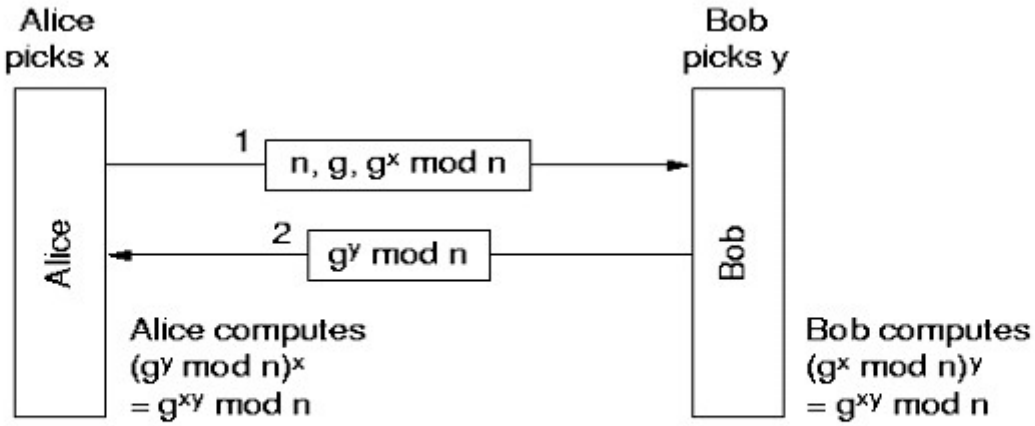


Fig. 7-15. The Diffie-Hellman key exchange.

Рис. 7-15.

А шлет В сообщение (n,g,g^x modn), В шлет в ответ (g^y modn). Теперь А выполняет операцию (g^y modn) ^{x}, В – (g^x modn) ^{y}. Удивительно, но теперь оба имеют общий ключ - g^{xy} modn! Например, n=47, g=3, x=8, y=10, то А шлет В сообщение (47,3,28), поскольку 3^8 mod47 = 28. В шлет А (17). А вычисляет 17^8 mod47 = 4, В вычисляет 28^{10} mod47 = 4. Ключ установлен – 4!

Злоумышленник следит за всем этим. Единственно, что мешает ему вычислить x и y – это то, что не известно алгоритма с приемлемой сложностью для вычисления логарифма от модуля для простых чисел. Однако, у этого алгоритма есть слабое место. **Рис.7-16** показывает его. Такой прием называется ***чужой в середине***.

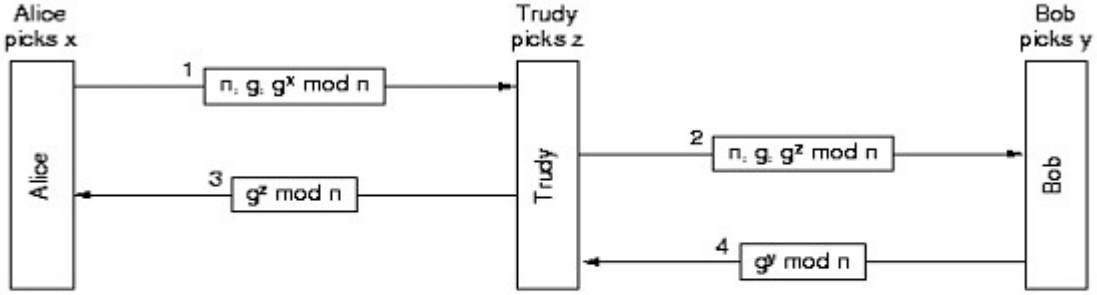


Fig. 7-16. The bucket brigade attack.

Рис. 7-16.

Проверка подлинности через центр раздачи ключей.

Договариваться с незнакомцем об общем секрете можно, но вряд ли это следует делать сразу (атака не спелого винограда). Кроме этого, общение с n людьми потребует хранения n ключей, что для общительных или популярных личностей может быть проблемой.

Другое решение можно получить, введя надежный центр распространения ключей (KDC). Его использование иллюстрирует **рис.7-17**.

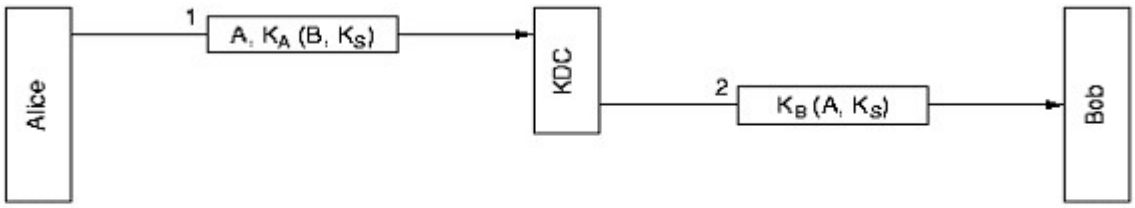


Fig. 7-17. The wide-mouth frog authentication protocol.

Рис. 7-17.

Идея этого протокола состоит вот в чем. А выбирает ключ сессии K^S. Используя свой ключ K^A, шлет в центр KDC запрос на соединение с В. Центр KDC, знает В и его ключ K^B. С помощью этого ключа KDC сообщает В ключ сессии K^S и кто хочет с ним с соединиться.

Однако, решение с центром KDC имеет изъян. Пусть злоумышленник как-то убедил А связаться с В и скопировал весь обмен сообщениями. Позже он может воспроизвести этот обмен за А и заставить В действовать так, как если бы с В говорил А! Этот способ атаки называется ***атака подменой***.

Против такой атаки есть несколько решений. Одно из них – ***временные метки***. Однако, это решение требует синхронизации часов. Поскольку в сети всегда есть расхождение в показаниях часов, то надо будет дать

определенный допуск, интервал, в течении которого считать сообщений верным. Злоумышленник может использовать приемом атаки подменой в течении этого интервала.

Другое решение использование ***разовых меток***. Однако. каждая из сторон должна помнить все разовые метки, использованные ранее. Это обременительно. Кроме этого, если список использованных разовых меток будет утерян по каким-либо причинам, то весь метод перестанет работать. Можно комбинировать решения разовых меток и временных меток.

Более тонкое решение установления подлинности дает многосторонний вызов-ответ протокол. Хорошо известным примером такого протокола является протокол Нидхема-Шредера, вариант которого показан на **рис.7-18**. В начале А сообщает KDC, что он хочет взаимодействовать с В. KDC сообщает ключ сессии, разовую метку R_A, шифруя сообщение ключом А. Разовая метка защищает А от подмены. Теперь, имея ключ сессии, А начинает обмен сообщениями с В. R_{A2} и R_B – разовые метки, защищающие А и В от подмен.

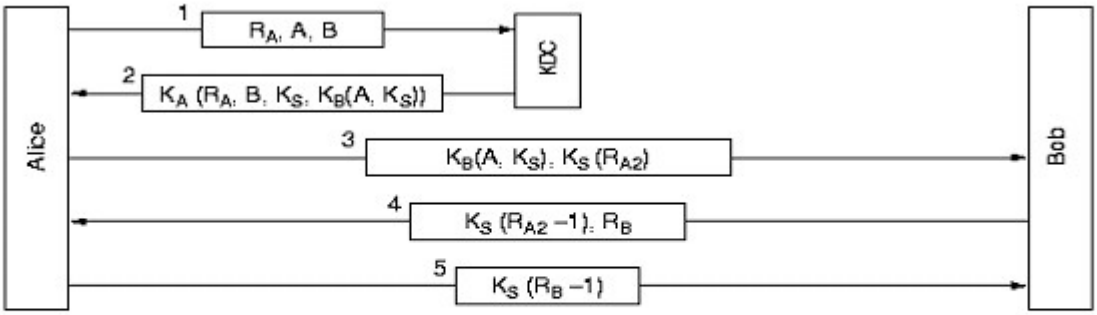


Fig. 7-18. The Needham-Schroeder authentication protocol.

Рис. 7-18.

Хотя этот протокол в целом надежен, но все-таки есть небольшая опасность. Если злоумышленник раздобудет все-таки старый ключ сессии, то он сможет подменить сообщение 3 старым и убедить В, что это А! На **рис.7-19** приведена схема исправленного протокола, предложенного Отвей и Рисом. В этой модификации KDC следит, чтобы R было одним и тем же в обеих частях сообщения 2.

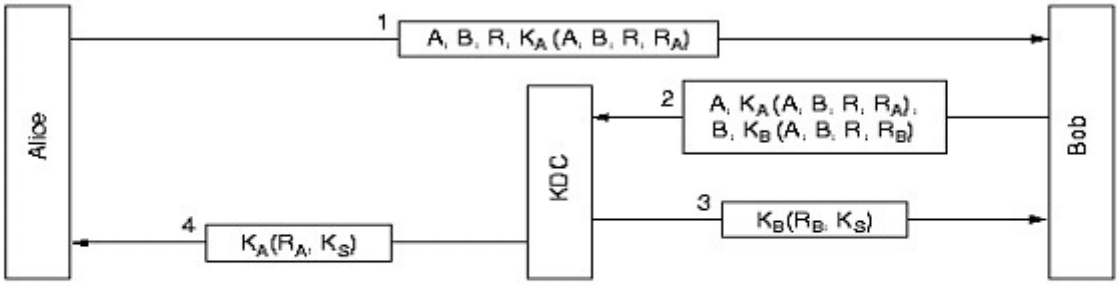


Fig. 7-19. The Otway-Rees authentication protocol (slightly simplified).

Рис. 7-19.

Установление подлинности протоколом Цербер.

Протокол установления подлинности Цербер используется многими практически действующими системами. Он представляет собой вариант протокола Нидхема-Шредера и был разработан в MIT для безопасного доступа в сеть (предотвратить несанкционированное использование ресурсов сети). В нем существенно использовано предположение, что все часы в сети хорошо синхронизованы.

Протокол Цербер предполагает использование кроме рабочей станции А еще трех серверов:

- Сервер установления подлинности (СП) – проверяет пользователей на этапе login;
- Сервер выдачи билета (СВБ) – идентификация билетов;

- Сервер B – тот кто должен выполнить работу, необходимую A.

СП аналогичен KDC и знает секретный пароль для каждого пользователя. СВБ выдает билеты, которые подтверждают подлинность заказчиков работ.

На **рис.7-20** показана работа протокола Цербер. Сначала пользователь садится за рабочую станцию и шлет открыто свое имя серверу СП. СП отвечает ключом сессии и билетом – KS, K^{TGS} (A,K^{S}). Все это зашифровано секретным ключом A. Когда сообщение 2 пришло на рабочую станцию у A запрашивают пароль, чтобы по нему установить K^{A}, для расшифровки сообщения 2. Пароль перезаписывается с временной меткой, чтобы предотвратить его захват злоумышленником. Выполнив login, пользователь может сообщить станции, что ему нужен сервер B. Рабочая станция обращается к СВБ за билетом для использования сервера B. Ключевым элементом этого запроса является K^{TGS} (A,K^{S}), зашифрованное секретным ключом СВБ. В ответ СВБ шлет ключ для работы A и B - K^{AB}.

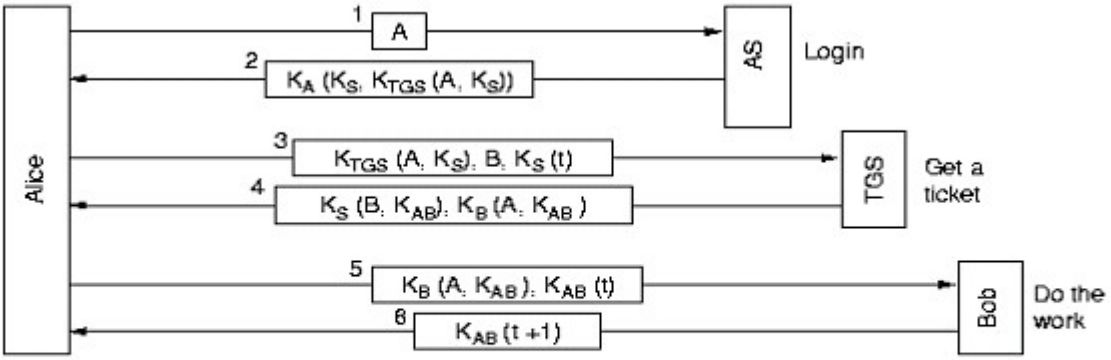


Fig. 7-20. The operation of Kerberos V4.

Рис. 7-20.

Теперь A может обращаться непосредственно к B с этим ключом. Это взаимодействие сопровождается временными метками, чтобы защититься от подмены. Если позднее A понадобиться работать с сервером C, то A должен будет повторить сообщение 3, но указать там сервер C.

Поскольку сеть может быть очень большой, то нельзя требовать, чтобы все использовали один и тот же СП. Сеть разбивают на области, в каждой свои СП и СВБ, которые взаимодействую между собой.

Установление подлинности, используя шифрование с открытым ключом

Установить взаимную подлинность можно с помощью шифрования с открытым ключом. Пусть A и B уже знают открытые ключи друг друга. Они их используют, чтобы установить подлинность друг друга, а затем использовать шифрование с секретным ключом, которое на несколько порядков быстрее.

На **рис.7-21** показана схема установления подлинности с шифрованием открытыми ключами. Здесь R^{A} и R^{B} используются, чтобы убедить A и B в их подлинности. Единственным слабым местом этого протокола является предположение, что A и B уже знают открытые ключи друг друга. Обмен такими ключами уязвим для атаки типа чужой в середине.

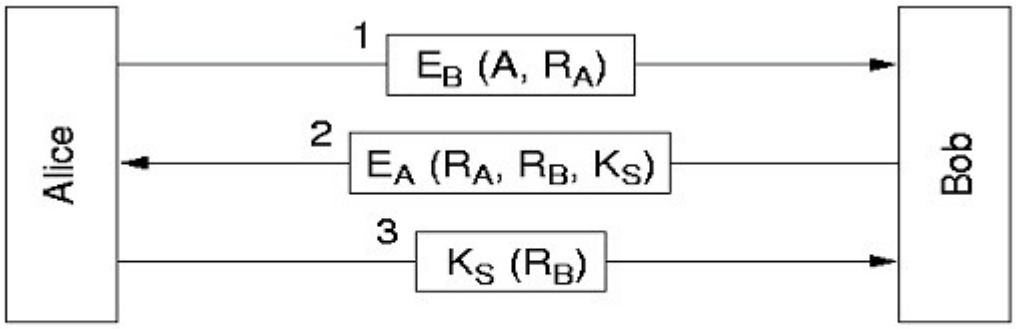


Fig. 7-21. Mutual authentication using public-key cryptography.

Рис. 7-21.

Ривст и Шамир предложили протокол защищенный от атаки чужой в средине. Это, так называемый, протокол с внутренним замком. Его идея передавать сообщения в два этапа: сначала только четные биты, затем нечетные.

7.1.6. Электронная подпись

Подлинность многих юридических, финансовых и прочих документов устанавливается наличием подписи уполномоченного лица. Поскольку есть способы отличить фотокопии от подлинника, то фотокопии не рассматриваются. Такая же проблема возникает для документов в электронной форме.

Проблема электронного аналога для ручной подписи весьма сложна. Нужна система, которая позволяла одной стороне посылать «подписанный» документ другой стороне так, чтобы

1. Получатель мог удостовериться в подлинности отправителя;
2. Отправитель позднее не мог отречься от документа;
3. Получатель не мог подделать документ.

Первое требование важно, например, при взаимодействии с банком, чтобы убедиться, что тот кто проводит операцию, действительно есть владелец счета.

Второе требование – клиент запросил закупить тонну золота, цена которого на бирже неожиданно упала. У клиента может возникнуть соблазн отказаться от своей заявки.

Третье требование предотвращает ситуации типа: цена на золото в предыдущем примере неожиданно подскочила, тогда у банка может появиться соблазн изобразить, что клиент просил купить не тонну, а, скажем, килограмм золота.

Подпись с секретным ключом

Одно из решений проблемы электронной подписи – наделить полномочиями третью сторону, которую знают все, которая знает всех и которой верят все. Назовем ее Сердечный Друг (СД). На **рис.7-22** показана схема такого решения. Что произойдет если А позже откажется от посланного сообщения? В суде В предъявит

сообщение А, K^{BB} (A,t,P), которое будет отправлено СД, как непререкаемому авторитету. СД расшифрует своим ключом эту запись и все увидят A,t,P.

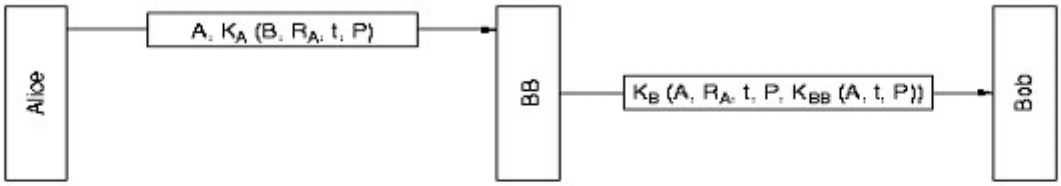


Fig. 7-22. Digital signatures with Big Brother.

Рис. 7-22.

Единственная слабость такого решения – злоумышленник может скопировать диалог между отправителем и получателем через СД и позже его повторить. Механизм временных меток позволяет уменьшить эту проблему. Кроме этого сохранение последних RA позволяет В заметить их повторное использование.

Подпись на основе открытого ключа.

Недостаток вышеописанного решения в том, что все должны доверять СД, который может читать сообщения. Кандидатами на его роль может быть правительство, банк, нотариус. Однако, далеко не все испытывают доверие к этом организациям.

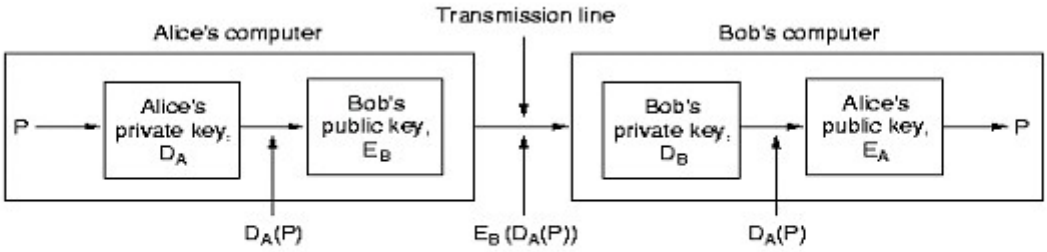


Fig. 7-23. Digital signatures using public-key cryptography.

Рис. 7-23.

На **рис.7-23** показа схема электронной подписи на основе открытых ключей. Здесь:

- D^X - закрытый ключ, E^X – открытый ключ;
- Предполагаем E(D(P))=P дополнительно к D(E(P))=P (этим свойством обладает RSA);

Здесь есть два недостатка. Оба основаны на том, что схема работает до тех пор, пока сторона A либо умышленно не рассекретила свой ключ, либо не изменила его в одностороннем порядке. Судебный случай: B предъявляет P и D^A(P), так как он не знает E^A, то он не мог подделать D^A(P). При этом, должно быть E^A(D^A(P))=P, в чем суд легко может убедиться.

Если A обращается в суд, то есть P и E^A(P), что легко сопоставить с тем, что есть у B. Однако, если A заявит, что у него украли ключи, а сам тайно передаст их, либо сменит их, не сообщив об этом B. В последнем случае текущий E^A будет не применим к тому D^A(P), который предъявит B. Здесь надо сопоставлять даты передачи сообщения и смены ключей.

7.4.5. Конфиденциальность почты

Пославший почту, естественно предполагает, что ее никто не читает кроме адресата. Однако, если об этом специально не позаботиться, то гарантировать этого нельзя. Далее мы рассмотрим две широко распространенных безопасных почтовых системы PGP и PEM.

PGP – Pretty Good Privacy

PGP – вполне хорошая конфиденциальность – разработка одного человека Фила Зиммермана (Phil Zimmermann 1995). Это полный пакет безопасности, который включает средства конфиденциальности, установления подлинности, электронной подписи, сжатия и все это в удобной для использования форме. Благодаря этому, а также что это разработка далекого от государственных структур человека, качественная, работает как на платформе Unix, так и MS-DOS/Windows, Macintosh и распространяется бесплатно, этот она получила очень широкое распространение.

Зиммерман был обвинен в нарушении ряда законов США о шифровании. Это позволило ему выдвинуть лозунг «Если конфиденциальность – вне закона, то она доступна только тем, кто вне закона».

PGP использует алгоритмы шифрования RSA, IDEA и MD5. PGP поддерживает компрессию, передаваемых данных, их секретность, электронную подпись и средства управления доступа к ключам. Схема работы PGP показана на **рис.7-49**. На этом рисунке – DA , DB личные (закрытые) ключи A и B соответственно, а EA , EB – их открытые ключи. Отметим, что секретный ключ для IDEA строиться автоматически по ходу работы PGP на стороне A и называется ключом сессии - KM, который затем шифруется алгоритмом RSA с открытым ключом пользователя B. Так же следует обратить внимание на то, что медленный алгоритм RSA используется для шифрования коротких фрагментов текста: 128 бит MD5 и 128 бит IDEA ключа.

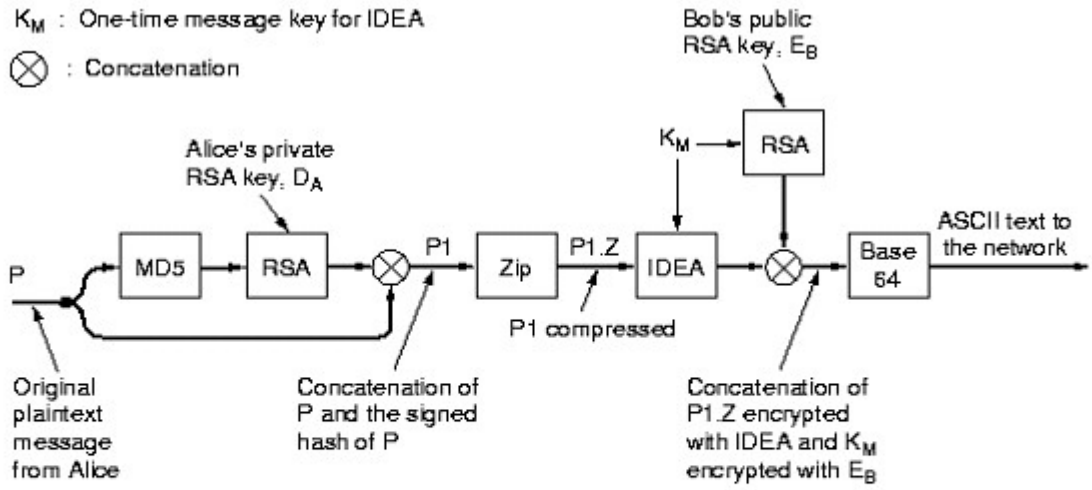


Fig. 7-49. PGP in operation for sending a message.

Рис. 7-49.

PGP поддерживает три длины ключей:

- Обычный – 314 бит (может быть раскрыт за счет больших затрат);
- Коммерческий – 512 бит (может быть раскрыт специализированными организациями, названия которых, как правило, состоит из трех букв);
- Военный – 1024 бита (не может быть раскрыт пока ни кем на земле).

Формат PGP сообщения показан на **рис.7-50**.

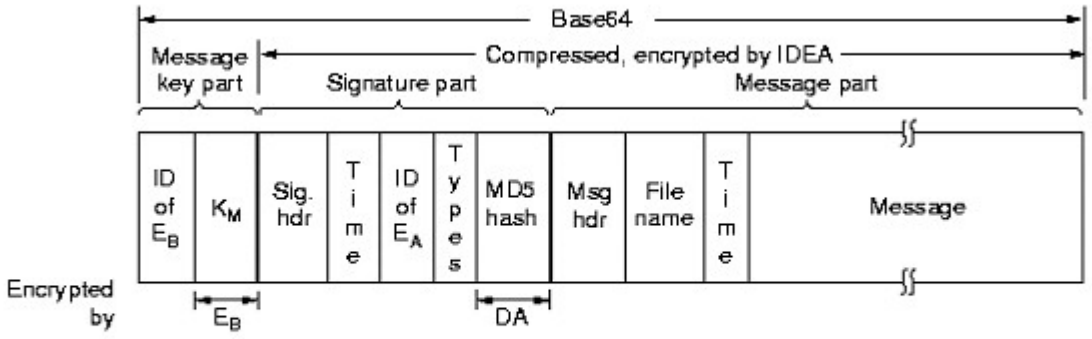


Fig. 7-50. A PGP message.

Рис. 7-50.

PEM – почтовая служба с повышенной конфиденциальность

PEM – имеет статус Internet стандарта (RFC 1421, 1424). Сообщения, пересылаемые с помощью PEM, сначала преобразуют в каноническую форму. В этой форме соблюдены соглашения относительно спецсимволов типа табуляции, последовательных пробелов и т.п. Затем сообщение обрабатывается MD5 или MD2, шифруется с помощью DES (56 разрядный ключ) и передается с помощью base64 кодировки. Передаваемый ключ защищается либо с помощью RSA, либо с помощью DES по схеме EDE.

На **рис.7-51** дано сравнение этих двух почтовых систем.

Item	PGP	PEM
Supports encryption?	Yes	Yes
Supports authentication?	Yes	Yes
Supports nonrepudiation?	Yes	Yes
Supports compression?	Yes	No
Supports canonicalization?	No	Yes
Supports mailing lists?	No	Yes
Uses base64 coding?	Yes	Yes
Current data encryption algorithm	IDEA	DES
Key length for data encryption (bits)	128	56
Current algorithm for key management	RSA	RSA or DES
Key length for key management (bits)	384/512/1024	Variable
User name space	User defined	X.400
X.509 conformant?	No	Yes
Do you have to trust anyone?	No	Yes (IPRA)
Key certification	Ad hoc	IPRA/PCA/CA hierarchy
Key revocation	Haphazard	Better
Can eavesdroppers read messages?	No	No
Can eavesdroppers read signatures?	No	Yes
Internet Standard?	No	Yes
Designed by	Small team	Standards committee

Fig. 7-51. A comparison of PGP and PEM.

Рис. 7-51.

Источник: Курс смелянского

22. Унифицированный язык моделировании UML. Основные средства языка.

Цели, история создания и назначение UML

Унифицированный язык моделирования UML (Unified Modeling Language) – это преемник того поколения методов объектно-ориентированного анализа и проектирования, которые появились в конце 80-х и начале 90-х годов. Создание UML фактически началось в конце 1994 г., когда Гради Буч и Джеймс Рамбо начали работу по объединению их методов под эгидой компании Rational Software. К концу 1995 г. они создали первую спецификацию объединенного метода, названного ими Unified Method, версия 0.8. Тогда же в 1995 г. к ним присоединился создатель метода OOSE (Object-Oriented Software Engineering) Ивар Якобсон. Таким образом, UML является прямым объединением и унификацией методов Буча, Рамбо и Якобсона, однако дополняет их новыми возможностями.

UML находится в процессе стандартизации, проводимом консорциумом OMG (Object Management Group), в настоящее время он принят в качестве стандартного языка моделирования и получил широкую поддержку. UML принят на вооружение практически всеми крупнейшими компаниями – производителями программного обеспечения (Microsoft, IBM, Hewlett-Packard, Oracle, Sybase и др.). Кроме того, практически все мировые производители CASE-средств, помимо Rational Software (Rational Rose), поддерживают UML в своих продуктах (Paradigm Plus (CA), System Architect (Popkin Software), Microsoft Visual Modeler и др.).

Состав диаграмм UML и их назначение

UML представляют его как язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов. Стандарт UML версии 1.1, принятый OMG в 1997 г., предлагает следующий набор диаграмм для моделирования:

Диаграммы вариантов использования

Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования описывает типичное взаимодействие между пользователем и системой. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать.

Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. Действующие лица представляют собой роли, а не конкретных людей или наименования работ. Несмотря на то, что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой, которой необходима некоторая информация от данной системы. Показывать на диаграмме действующих лиц следует только в том случае, когда им действительно необходимы некоторые варианты использования.

Действующие лица делятся на три основных типа – пользователи системы, другие системы, взаимодействующие с данной, и время. Время становится действующим лицом, если от него зависит запуск каких-либо событий в системе.

Диаграммы взаимодействия

Диаграммы взаимодействия (interaction diagrams) описывают поведение взаимодействующих групп объектов.

Сообщение (message) – это средство, с помощью которого объект-отправитель запрашивает у объекта получателя выполнение одной из его операций.

Информационное (informative) сообщение – это сообщение, снабжающее объект-получатель некоторой информацией для обновления его состояния.

Сообщение-запрос (interrogative) – это сообщение, запрашивающее выдачу некоторой информации об объекте-получателе.

Императивное (imperative) сообщение – это сообщение, запрашивающее у объекта-получателя выполнение некоторых действий.

Существует два вида диаграмм взаимодействия: диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams).

На диаграмме последовательности объект изображается в виде прямоугольника, от которого вниз проведена пунктирная вертикальная линия. Эта линия называется линией жизни (lifeline) объекта. Она представляет собой фрагмент жизненного цикла объекта в процессе взаимодействия.

Подобно диаграммам последовательности, кооперативные диаграммы (collaborations) отображают поток событий через конкретный сценарий варианта использования. Диаграммы последовательности упорядочены по времени, а кооперативные диаграммы больше внимания заостряют на связях между объектами.

Диаграммы классов

Диаграмма классов определяет типы классов системы и различного рода статические связи, которые существуют между ними. На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами.

Стереотипы – это механизм, позволяющий разделять классы на категории. В языке UML определены три основных стереотипа классов: Boundary (граница), Entity (сущность) и Control (управление).

Диаграммы состояний
Диаграммы состояний определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий.

Диаграммы деятельности
В отличие от большинства других средств UML, диаграммы деятельностей не имеют явно выраженного источника в предыдущих работах Буча, Рамбо и Якобсона, и заимствуют идеи из нескольких различных методов, в частности, метода моделирования состояний SDL и сетей Петри. Эти диаграммы особенно полезны в описании поведения, включающего большое количество параллельных процессов.

Самым большим достоинством диаграмм деятельностей является поддержка параллелизма. Благодаря этому они являются мощным средством моделирования потоков работ и, по существу, параллельного программирования. Самый большой их недостаток заключается в том, что связи между действиями и объектами просматриваются не слишком четко.

Диаграммы компонентов
Диаграммы компонентов показывают, как выглядит модель на физическом уровне. На них изображены компоненты программного обеспечения и связи между ними. При этом на такой диаграмме выделяют два типа компонентов: исполняемые компоненты и библиотеки кода.

Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию системы. Из нее видно, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут созданы системой. На такой диаграмме показано соответствие классов реализованным компонентам. Она нужна там, где начинается генерация кода.

Диаграммы размещения
Диаграмма размещения (deployment diagram) отражает физические взаимосвязи между программными и аппаратными компонентами системы. Она является хорошим средством для того, чтобы показать маршруты перемещения объектов и компонентов в распределенной системе.

Каждый узел на диаграмме размещения представляет собой некоторый тип вычислительного устройства – в большинстве случаев, часть аппаратуры. Эта аппаратура может быть простым устройством или датчиком, а может быть и мэйнфреймом.

Диаграмма размещения используется менеджером проекта, пользователями, архитектором системы и эксплуатационным персоналом, чтобы понять физическое размещение системы и расположение её отдельных подсистем.

23. Основы программной инженерии. Каскадная и итерационная модели жизненного цикла программного обеспечения.

Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ и методы разработки ПО (под моделью ЖЦ понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении ЖЦ. Модель ЖЦ зависит от специфики ИС и специфики условий, в которых последняя создается и функционирует). Его регламенты являются общими для любых моделей ЖЦ, методологий и технологий разработки. Стандарт ISO/IEC 12207 описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.
К настоящему времени наибольшее распространение получили следующие две основные модели ЖЦ:

- каскадная модель (70-85 г.г.);
- спиральная модель (86-90 г.г.).

В изначально существовавших однородных ИС каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся каскадный способ. Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем (рис. 1.1). Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.
Положительные стороны применения каскадного подхода заключаются в следующем [2]:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

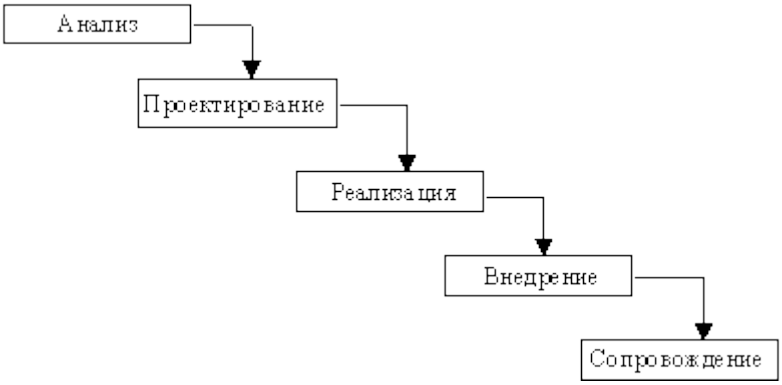


Рис. 1.1. Каскадная схема разработки ПО

Каскадный подход хорошо зарекомендовал себя при построении ИС, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако, в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал следующий вид (рис. 1.2):

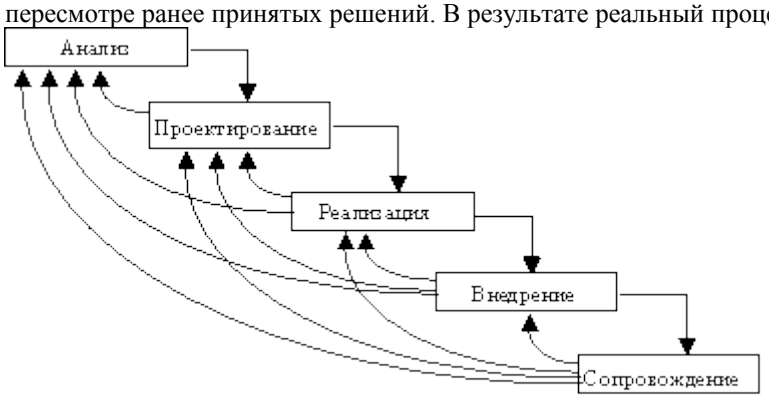


Рис. 1.2. Реальный процесс разработки ПО по каскадной схеме

Основным недостатком каскадного подхода является существенное запаздывание с получением результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к ИС "заморожены" в виде технического задания на все время ее создания. Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО, пользователи получают систему, не удовлетворяющую их потребностям. Модели (как функциональные, так и информационные) автоматизируемого объекта могут устареть одновременно с их утверждением.

Для преодоления перечисленных проблем была предложена спиральная модель ЖЦ [10] (рис. 1.3), делающая упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации.

Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла - определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

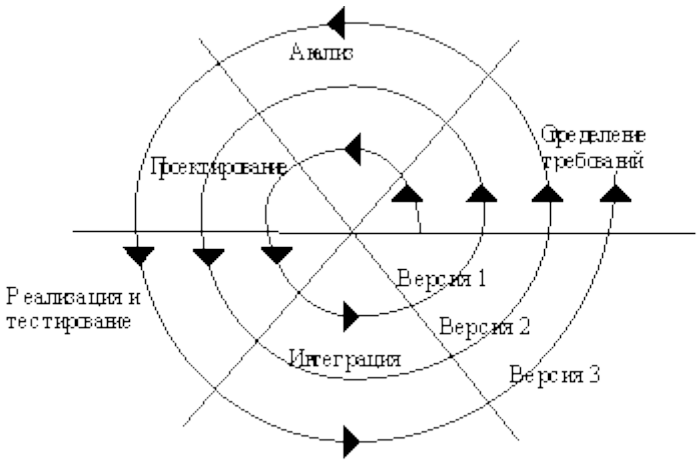
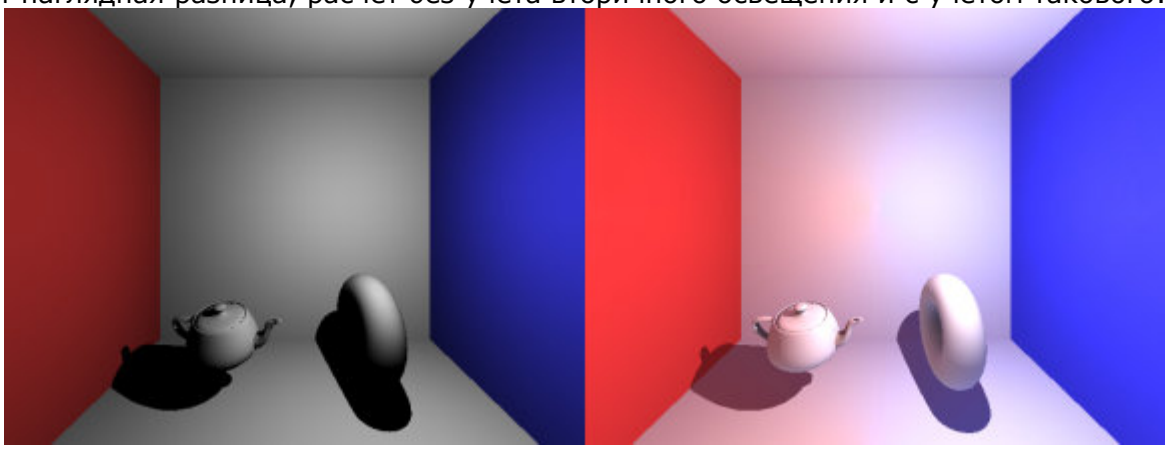


Рис 1.3. Спиральная модель ЖЦ

24. Глобальные и локальные модели освещения в компьютерной графике. Модель Фонга.

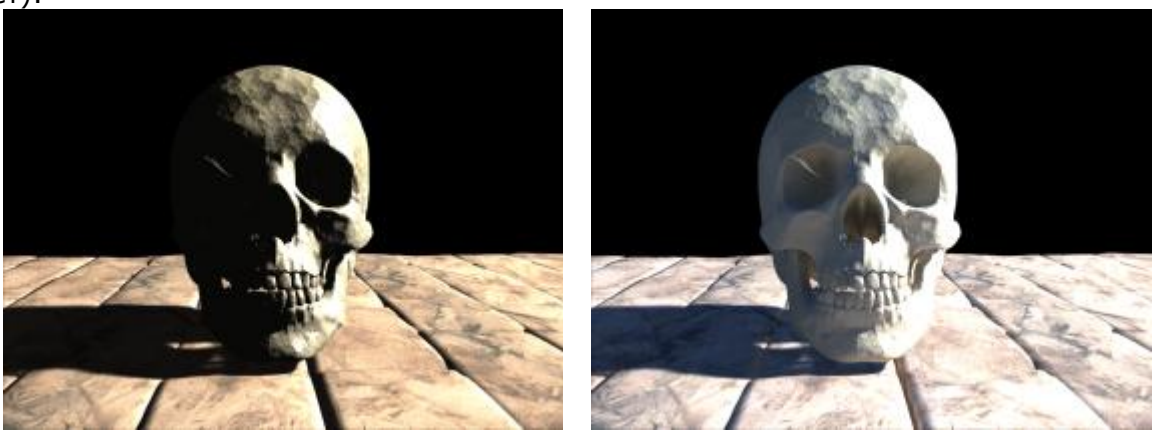
Реалистичное освещение сцены смоделировать сложно, каждый луч света в реальности многократно отражается и преломляется, число этих отражений не ограничено. А в 3D рендеринге число отражений сильно зависит от расчетных возможностей, любой расчет сцены является упрощенной физической моделью, а получаемое в итоге изображение лишь приближено к реалистичности.
Алгоритмы освещения можно разделить на две модели: прямое или локальное освещение и глобальное освещение (direct или local illumination и global illumination). Локальная модель освещения использует расчет прямой освещенности, свет от источников света до первого пересечения света с непрозрачной поверхностью, взаимодействие объектов между собой не учитывается. Хотя такая модель пытается компенсировать это добавлением фонового или равномерного (ambient) освещения, но это самая простая аппроксимация, сильно упрощенное освещение от всех непрямых лучей источников света, которое задает цвет и интенсивность освещения объектов в отсутствии прямых источников света.
Той же трассировкой лучей рассчитывается освещенность поверхностей только прямыми лучами от источников света и любая поверхность, для того, чтобы быть видимой, должна быть напрямую освещена источником света. Этого недостаточно для достижения фотореалистичных результатов, кроме прямого освещения нужно учитывать и вторичное освещение отраженными от других поверхностей лучами. В реальном мире лучи света отражаются от поверхностей несколько раз, пока не затухнут совсем. Солнечный свет, проходящий через окно, освещает всю комнату целиком, хотя лучи не могут напрямую достигать всех поверхностей. Чем ярче источник света, тем большее количество раз он будет отражаться. Цвет отражающей поверхности также влияет на цвет отраженного света, например, красная стена вызовет красное пятно на соседнем объекте белого цвета. Вот наглядная разница, расчет без учета вторичного освещения и с учетом такового:



В глобальной модели освещения, global illumination, рассчитывается освещение с учетом влияния объектов друг на друга, учитываются многократные отражения и преломления лучей света от поверхностей объектов, каустика (caustics) и подповерхностное рассеивание (subsurface scattering). Эта модель позволяет получить более реалистичную картинку, но усложняет процесс, требуя заметно больше ресурсов. Существует несколько алгоритмов global illumination, мы вкратце рассмотрим radiosity (расчет непрямого освещения) и photon mapping (расчет глобального освещения на основе карт фотонов, предрассчитанных при помощи трассировки). Есть и

упрощенные методы по симуляции непрямого освещения, такие, как изменение общей яркости сцены в зависимости от количества и яркости источников света в ней или использование большого количества точечных источников света, расставленных по сцене для имитации отраженного света, но все же это далеко от настоящего алгоритма GI.
Алгоритм radiosity это процесс расчета вторичных отражений лучей света от одних поверхностей к другим, а также, от окружающей среды к объектам. Лучи от источников света трассируются до тех пор, пока сила их не снизится ниже определенного уровня или лучи достигнут определенного числа отражений. Это распространенная техника GI, вычисления обычно выполняются перед визуализацией, а результаты расчета можно использовать для рендеринга в реальном времени. Основные идеи radiosity основаны на физике теплового переноса. Поверхности объектов разбиваются на небольшие участки, называемые патчами, и принимается, что отраженный свет рассеивается равномерно во все стороны. Вместо расчета каждого луча для источников света, используется техника усреднения, разделяющая источники света на патчи, основываясь на уровнях энергии, которые они выдают. Эта энергия распределяется между патчами поверхностей пропорционально.
Еще один метод расчета глобальной освещенности предложен Henrik Wann Jensen, это метод фотонных карт photon mapping. Использование фотонных карт - это другой алгоритм расчета глобального освещения, основанный на трассировке лучей и используемый для имитации взаимодействия лучей света с объектами сцены. Алгоритмом рассчитываются вторичные отражения лучей, преломление света через прозрачные поверхности, рассеянные отражения. Этот метод состоит в расчете освещенности точек поверхности в два прохода. В первом выполняется прямая трассировка лучей света с вторичными отражениями, это предварительный процесс, выполняемый перед основным рендерингом. В этом методе рассчитывается энергия фотонов, идущих от источника света к объектам сцены. Когда фотоны достигают поверхности, точка пересечения, направление и энергия фотона сохраняются в кэш, называемый фотонной картой. Фотонные карты могут сохраняться на диске для последующего использования, чтобы не просчитывать их каждый кадр. Отражения фотонов просчитываются до тех пор, пока работа не останавливается после определенного количества отражений или при достижении определенной энергии. Во втором проходе рендеринга выполняется расчет освещения пикселей сцены прямыми лучами, с учетом данных, сохраненных в фотонных картах, энергия фотонов добавляется к энергии прямого освещения.
Расчеты global illumination, использующие большое количество вторичных отражений, занимают гораздо больше времени, чем расчеты прямого освещения. Существуют техники для аппаратного расчета радиосити в реальном времени, которые используют возможности программируемых видеочипов последних поколений, но пока что сцены, для которых рассчитывается глобальное освещение в реальном времени, должны быть достаточно простыми и в алгоритмах делается множество упрощений.
Но вот что давно используют - так это статическое предпросчитанное глобальное освещение, что приемлемо для сцен без изменения положения источников света и больших объектов, сильно влияющих на освещение. Ведь расчет глобального освещения не зависит от положения наблюдателя и если в сцене не изменяется положение таких объектов сцены и параметров источников освещения, то можно использовать заранее просчитанные значения освещенности. Это используют во многих играх, сохраняя данные GI расчетов в виде лайтмапов (lightmaps).
Существуют и приемлемые алгоритмы для имитации глобального освещения в динамике. Например, есть такой простой метод для использования в приложениях реального времени, для расчета непрямого освещения объекта в сцене: упрощенный рендеринг всех объектов с пониженной детализацией (за исключением того, для которого считают освещение), в кубическую карту низкого разрешения (ее также можно использовать для отображения динамических отражений на поверхности объекта), затем фильтрация этой текстуры (несколько проходов blur фильтра), и применение для освещения этого объекта данных из рассчитанной текстуры в качестве дополнения к прямому освещению. В случаях, когда динамический расчет слишком тяжел, можно обойтись статическими radiosity картами. Пример из игры MotoGP 2, на котором хорошо видно благотворное влияние даже такой простой имитации GI:

Напоследок еще один пример direct illumination против global illumination рендеринга, чтобы развеять оставшиеся сомнения о полезности вторичного освещения (источник света один, ambient освещение отсутствует):



Формула освещенности Фонга

Рассмотрим, как модель Фонга вписывается в общий случай. Вспомним, формула Фонга определяет количество света, отраженного в сторону наблюдателя следующим образом:

$$L_o = L_i\left(k_d(\mathbf{L}\bullet\mathbf{N})+k_s(\mathbf{R}\bullet\mathbf{V})^n\right)$$

где

- L – направление на источник света;
- V – направление на наблюдателя
- N – нормаль к поверхности;
- R – направление отражения.
- k_d и k_s – т.н. диффузный и зеркальный коэффициенты, используетюся для управления вкладом диффузного и зеркального отражения ;
- n – коэффициент для изменения ширины зеркального блика

Модель Фонга – локальная, т.е. она не учитывает вторичного освещения. Поэтому часто в нее включают коэффициент, отвечающий за *фоновое (*ambient) освещение для аппроксимации вторичного освещения. Без ограничений общности, мы не будем рассматривать этот коэффициент. Обозначим выражение в скобках через *Refl*(**L**,**V**), получим:

$$L_o = L_i Refl(\mathbf{L},\mathbf{V})$$

L и **V** соответствуют входящему направлению w_i и, соответственно, исходящему направлению w_o в терминологии ДФОС. Перепишем выражение следующим образом:

$$L_o = L_i Refl(w_i, w_o)$$

$$L_o = Refl(w_i, w_o)L_i$$

$$L_o = \frac{\cos\theta_i dw_i}{\cos\theta_i dw_i} Refl(w_i, w_o)L_i$$

$$L_o = \frac{Refl(w_i, w_o)}{\cos\theta_i dw_i} L_i \cos\theta_i dw_i$$

Это выражение имеет вид подынтегральной функции в общей формуле освещения для точечного источника света. Если рассматривать первый множитель как ДФОС, тогда модель освещения Фонга становится частным случаем общей модели освещения.

$$\mathrm{BRDF}(\theta_i, \phi_i, \theta_o, \phi_o) = \frac{Refl(w_i, w_o)}{\cos\theta_i dw_i}$$

$$\mathrm{BRDF}(\theta_i, \phi_i, \theta_o, \phi_o) = \frac{k_d(\boldsymbol{w}_i \cdot \mathbf{N}) + k_s(\mathbf{R} \cdot \boldsymbol{w}_o)^n}{\cos\theta_i dw_i}$$

Необходимо сделать несколько замечаний

- Хотя модель Фонга очень удобна для вычислений и ее вычисление реализовано в современной аппаратуре, на практике ей соответствует очень узкий класс материалов.
- Несмотря на то, что эта модель широко распространена, она не является физически корректной. В частности, для модели Фонга не выполняется закон сохранения энергии.

Аппроксимация света на модели Фонга

Существует несколько алгоритмов закраски сложных поверхностей. Методы закраски Гуро и Фонга являются наиболее популярными. При этом метод Фонга требует больших вычислительных затрат, однако он позволяет разрешить многие проблемы метода Гуро.

Модель Фонга

$I_s = I_l * w(\theta, \lambda) * \cos^n\alpha$, где

I_s — интенсивность света, попадающего в глаз наблюдателя,

I_l — интенсивность падающего луча,

w — коэффициент отражения, который находится из кривой отражения и зависит от угла падения θ и длины волны λ,

$\cos^n\alpha$ — характеристика материала поверхности,

α — угол, образованный лучами отражения и наблюдения,

n — узость освещающего луча.

Общая модель закраски

На **рис. 25.1** мы видим тело, как бы составленное из граней. По формулам из предыдущей лекции можно рассчитать освещенность для каждой точки этого тела. Но наблюдателю будут видны явно выраженные стыки поверхностей, поэтому для вуалирования переходов освещенности между этими поверхностями и проводят аппроксимацию (то есть сглаживание).

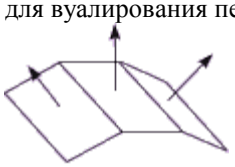


рис. 25.1

Закраска фигуры по Фонгу. Вуалирование граней

При закраске Фонга аппроксимация кривизны поверхности производится сначала в вершинах многоугольников — путем аппроксимации нормали в вершине. После этого билинейной интерполяцией вычисляется нормаль в каждом пикселе.

На **рис. 25.2** (а также на **рис. 25.3**) изображены четыре плоскости, спроецированные на экран монитора. Горизонтальная штрих-линия представляет собой горизонтальный ряд пикселов, который строится лучом ЭЛТ.

рис. 25.2

Рассмотрим метод Фонга, суть которого состоит в апроксимации нормалей, на примере.

1. Пусть нам известны уравнения плоскостей для трех многоугольников, и сходятся они в одной вершине. Три плоскости собираются в вершине V. Усредняем нормаль в ней:

   $n_v = (a_1 + a_2 + a_3)i + (b_1 + b_2 + b_3)j + (c_1 + c_2 + c_3)k$

   P_0: z - 1 = 0;

   P_1: -y + z - 2 = 0;

   P_2: -x + z - 2 = 0.

   n_v = -i - j + 3k — усредненная нормаль в точке *n* (используется для вычисления интенсивности пиксела).

$|n| = ((-1)^2 + (-1)^2 + 3^2)^{0.5} = 11^{0.5}$ — абсолютная величина нормали.

$e = n_v / |n| = -0.3i - 0.3j + 0.9k$ — единичная (нормированная) нормаль.

2. Вычисляем нормаль в каждом пикселе строки. Для закраски необходимы векторы нормали в точках A, B и C. Аппроксимируем их усреднением нормалей к окружающим плоскостям. Для того, чтобы изобразить объект методом построчного сканирования, нужно в соответствии с моделью освещения рассчитывать интенсивность каждого пиксела вдоль сканирующей строки. Сначала определяется интенсивность вершин многоугольника (A, B и C), а затем с помощью интерполяции вычисляется интенсивность каждого пиксела на сканирующей строке.

рис. 25.3

Нормаль в точке Q находится линейной интерполяцией между A и B:

$n_Q = U * n_A + (1 - U) * n_B$; $0 <= U <= 1$; $U = AQ/AB$.

Нормаль в точке R находится линейной интерполяцией между B и C:

$n_R = \omega * n_B + (1 - \omega) * n_C$; $\omega = BR/BC$.

Нормаль в точке P находится линейной интерполяцией между Q и R:

$n_P = t * n_Q + (1 - t) * n_R$; $0 <= t <= 1$; $t = QP/QR$.

Глава **1.** Локальная модель освещения **(local Illuminating model)**

В соответствии с принятым в компьютерной графике подходом, расчет освещенности распадается на две основные задачи. Первая – определить способ расчета освещенности в произвольной точке трехмерного пространства, решается при помощи построения обобщенной математической модели освещенности (Illuminating model). Вторая задача –применение Illuminating model для компьютерных расчетов освещенности трехмерных объектов с конкретной геометрией и свойствами поверхности, решается при помощи так называемой модели затенения (Shading model).

Моделей освещенности к настоящему моменту разработано несколько. Самая первая, и самая простая – локальная модель освещенности. Эта модель не рассматривает процессы светового взаимодействия объектов сцены между собой, а только расчет освещенности самих объектов. Вторая, глобальная модель освещенности – Global Illuminations, рассматривает трехмерную сцену как единую систему и пытается описывать освещение с учетом взаимного влияния объектов. В рамках этой модели рассматриваются такие вопросы, как многократное отражение и преломление света (ray tracing), рассеянное освещение (radiosity), caustic и subsurface scattering (photon mapping) и другие.

Начнем с самой простой модели освещенности, тем более что она и по сей день является главным способом расчета в рендерах scan-line типа (например, scan-line рендер 3ds max или mental ray).

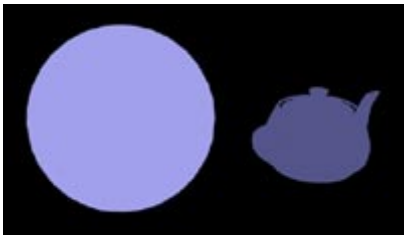
В рамках локальной модели освещения рассматривается свет только от явных точечных источников света трех типов – omni, spot и directional, а само взаимодействие ограничивается только однократным отражением света от непрозрачной поверхности. Изображение формируется в результате отражения падающего на поверхность объектов света, интенсивность и цвет которого и необходимо рассчитать.

В самом общем случае, в свете требования фотореалистичности, эта модель должна также учитывать и неявное ambient-освещение. Ambient-освещение, или его еще называют фоновым (background), – это окружающее объект освещение от удаленных источников, чье положение и характеристики не известны. Необходимость учета ambient-освещения, пусть и очень грубо, обусловлена тем, что его вклад может быть достаточно велик – до 50% от общей освещенности. В Local Illumination считают, что фоновое освещение задает цвет (и его интенсивность) объекта в отсутствии явных источников света или в тени. Не несет никакой информации об объекте, кроме значения простого цвета, равномерно заливающего контур объекта.

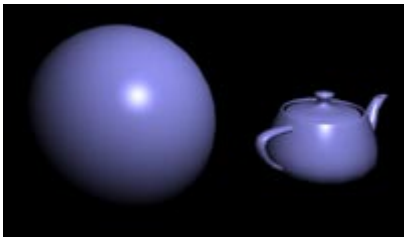
Интенсивность такого освещения постоянна и равномерно распределена во всем пространстве, расчет его отражения поверхностью выполняется по формуле:

$$I_{amb} = k_a \cdot I_a ,$$

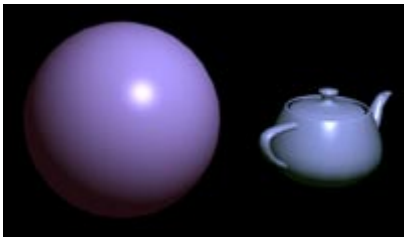
где I_{abm} – интенсивность отраженного *ambient* освещения, k_a – коэффициент, характеризующий отражающие свойства поверхности для *ambient*-освещения *(*его значение находится в пределах от *0* до *1),* I_a – исходная интенсивность *ambient*-света, падающего на поверхность.



Увидеть этот тип освещения в "чистом виде" в *3ds max* можно либо включив в настройках источника света свойство *ambient only*, либо изменив цвет *Ambient Light* панели *Environment* с черного (черный означает отсутствие окружающего освещения, принят по умолчанию) на более светлый и отключив *diffuse/specular/ambient* освещение в настройках источника света . В обычной ситуации, если цвет *Ambient light* в *Environment* отличен от черного *(0, 0, 0* в *RGB)*, свойство материалов *Ambient color* начинает оказывать влияние на цвет объекта – то есть *max* считает, что в сцене присутствует фоновое освещение, и начинает его просчитывать для поверхностей.

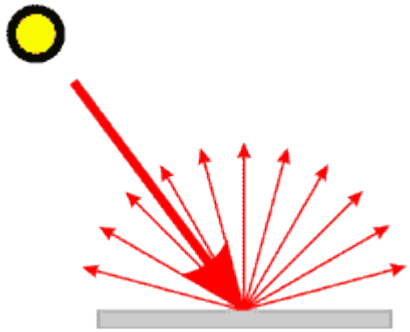


Объектам назначены материалы, абсолютно одинаковые, за исключением значения *ambient color.* При отключенном фоновом освещении они и выглядят одинаково.

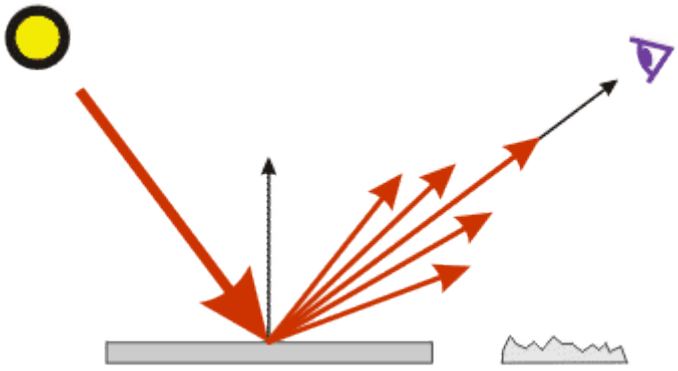


Включение фонового освещения в *Environment* проявляет различие в *ambient* характеристиках этих материалов.

Часть света от прямых источников зеркально отражается поверхностью, а остальной свет диффузно рассеивается во всех направлениях. Кроме чисто зеркального отражения, которое имеют идеально отполированные поверхности, различают так называемое glossiness или распределенное зеркальное отражение – отражение в некотором створе углов, а не на один единственный угол. Такое рассеяние света обусловлено микрорельефом ("шероховатостью") поверхности, то есть поверхность реальных объектов не является идеально гладкой, а состоит из большого количества микровыступов и впадин, которые зеркально отражают падающий свет под разными углами. Результатом glossy-отражения является specular highlight – яркий световой блик, имеющий размер в зависимости от степени шероховатости поверхности.



Диффузное рассеяние. Свет отражается равномерно под всеми углами.

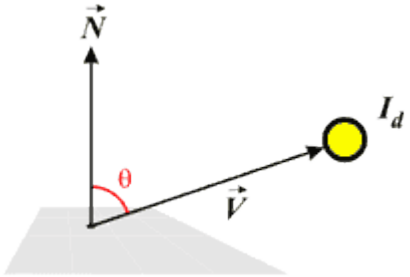


Зеркальное отражение под одним идеальным углом и *glossy* отражение в створе углов, обусловленное шероховатостью поверхности.

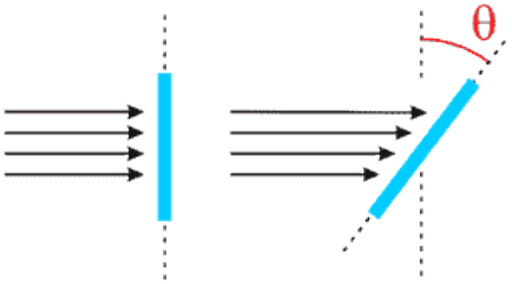
Интенсивность рассеянного света зависит от угла падающего на поверхность света по закону Ламберта (Lambert):

$$I_{diff} = I_d \cdot k_{diff} \cdot \cos(\theta),$$

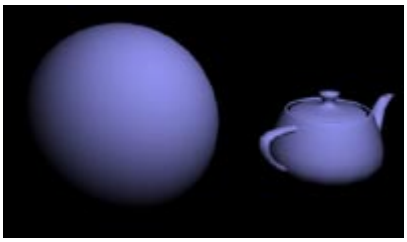
где I_d – интенсивность падающего на поверхность света, k_{diff} – коэффициент, характеризующий рассеивающие свойства поверхности *(*его значение изменяется в пределах от *0* до *1)*, $\cos(\theta)$ – угол между направлением на источник света и нормалью поверхности.



Другими словами, поверхность будет освещена больше, если свет падает на нее перпендикулярно $(\theta=0)$, и меньше, если свет падает под любым другим углом, поскольку в этом случае увеличивается освещаемая площадь.



Диффузно рассеянный свет является главным источником визуальной информации о геометрии трехмерных объектов.

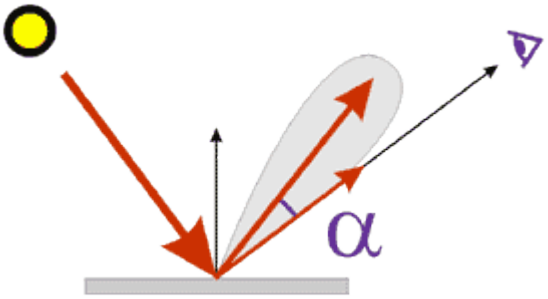


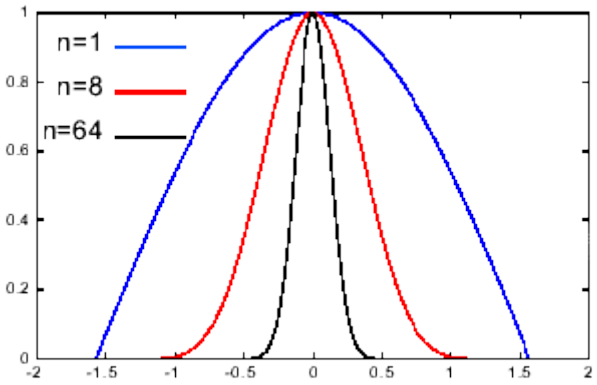
В *3ds max* диффузное отражение света можно увидеть, отключив *Specular* в свойствах источников света. Второй способ достичь того же результата – установить черный цвет для *Specular* в свойствах материала и *Ambient light* панели *Environment.* Полностью отключить диффузное отражение света для поверхности можно назначив черный цвет свойству материала *Diffuse.*

Как мы уже обсуждали, свет отражается зеркально в некотором створе углов, и для большинства реальных материалов мы всегда видим зеркальную подсветку в форме светового пятна, а не в форме яркой точки. Поэтому, для расчета интенсивности зеркально отраженного света используется формула, предложенная Фонгом:

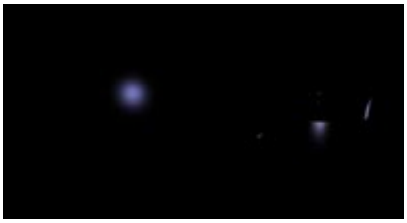
$$I_{spec} = I_s \cdot k_s \cdot \cos^{n}(\alpha),$$

где I_{spec} – интенсивность зеркально отраженного света, I_s – интенсивность источника света, k_s – коэффициент, характеризующий свойства зеркального отражения поверхности, α – угол между направлением идеального отражения и направлением на наблюдателя, степень ***n*** определяет размер пятна светового блика – чем больше ***n***, тем меньше световой блик, и тем ближе отражающие свойства поверхности к свойствам идеального зеркала.



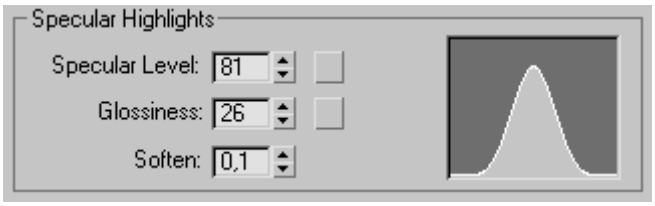


Формула Фонга – пример компьютерной фикции, поскольку она не имеет физического смысла. Ее используют просто потому, что она дает хорошие практические результаты.



Зеркальную подсветку в 3ds max можно увидеть, установив черный цвет Diffuse color материала и Ambient light панели Environment, либо включив только Specular в свойствах источников света. Для создания идеальной зеркально отражающей поверхности в свойствах материала необходимо полностью отключить диффузное отражение и ambient-цвет.

В 3ds max коэффициенту k_s соответствует параметр Specular Level, **n** – параметр Glossiness группы настроек материала Specular Highlights:



Наконец, чтобы различать объекты, находящиеся на разных расстояниях от камеры, используется функция затухания интенсивности света с расстоянием:

$$I(r) = I/r^2,$$

В компьютерной графике используется модифицированная формула, более приемлемая на практике:

$$I(r) = I/(a + br + cr^2),$$

Таким образом, локальная модель освещенности предполагает расчет отраженной фоновой освещенности, диффузного и зеркального отражения от прямых источников:

$$I_{полн}=I_{амб}+1/(a+br+cr^2)(I_{diff}+I_{spec})=k_a \bullet I_a+1/(a+br+cr^2)(I_d \bullet k_{diff} \bullet \cos(\theta)+I_s \bullet k_s \bullet \cos^n(\alpha))$$

Эта формула для расчета освещенности по одному каналу для одного источника света. Полная освещенность должна рассчитываться по всем трем основным каналам (RGB) и для всех источников освещения в трехмерной сцене и затем суммироваться. Эта модель получила название модели освещенности Фонга (не путать с моделью затенения Фонга, об этом ниже), по имени автора ее разработавшего. Является основной моделью в скан-лайн рендерах.

Как видите, локальная модель освещенности довольно проста. Проста настолько, что обсуждать ее достоинства и недостатки бессмысленно, она дает неплохие практические результаты – на том и остановимся.