

ПРОГРАММИРОВАНИЕ

1. Понятия алгоритм, программа, программирование. Процесс получения исполняемой программы из исходных кодов.

Алгоритм — набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий. В качестве исполнителя выступает некоторый механизм (компьютер, токарный станок, швейная машина).

Свойства алгоритмов:

- дискретность (получение однозначного результата при заданных исходных данных),
- детерминированность (алгоритм выдаёт один и тот же результат для одних и тех же исходных данных),
- понятность (каждый элементарный шаг известен исполнителю),
- конечность (при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов),
- универсальность (работоспособность на разных наборах данных), результативность (завершение с определенными результатами),
- корректность (алгоритм не содержит ошибок, если он даёт правильные результаты для любых допустимых исходных данных. Алгоритм содержит ошибки, если приводит к получению неправильных результатов либо не дает результатов вовсе).

Формы записи алгоритма:

- словесная или вербальная (языковая, формульно-словесная);
- псевдокод (формальные алгоритмические языки);
- графическая (блок-схемы).

Программа – 1) комбинация компьютерных инструкций и данных, позволяющая аппаратному обеспечению вычислительной системы выполнять вычисления или функции управления.

Программа – 2) синтаксическая единица, которая соответствует правилам определённого языка программирования, состоящая из определений и операторов или инструкций, необходимых для определённой функции, задачи или решения проблемы.

Программирование - написание инструкций (программ) на конкретном языке программирования (часто по уже имеющемуся алгоритму — плану, методу решения поставленной задачи)

Процесс получения исполняемой программы из исходных кодов:

Объединенная единым алгоритмом совокупность описаний и операторов образует программу на алгоритмическом языке. Для того чтобы выполнить программу, требуется перевести ее на язык, понятный процессору — в машинные коды. Этот процесс состоит из нескольких этапов.

Сначала программа передается препроцессору, который выполняет директивы, содержащиеся в ее тексте (например, включение в текст так называемых заголовочных файлов — текстовых файлов, в которых содержатся описания используемых в программе элементов).

Получившийся полный текст программы поступает на вход компилятора, который выделяет лексемы (последовательность допустимых символов языка программирования,

имеющая смысл для транслятора), а затем на основе грамматики языка распознает выражения и операторы, построенные из этих лексем. При этом компилятор выявляет синтаксические ошибки и в случае их отсутствия строит объектный модуль.

Компоновщик, или редактор связей, формирует исполняемый модуль программы, подключая к объектному модулю другие объектные модули, в том числе содержащие функции библиотек, обращение к которым содержится в любой программе (например, для осуществления вывода на экран). Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки.

2. Представление информации в ЭВМ. Системы счисления. Связь между системами счисления.

В ЭВМ применяется двоичная система счисления, т.е. все числа в компьютере представляются с помощью нулей и единиц, поэтому компьютер может обрабатывать только информацию, представленную в цифровой форме.

Для преобразования числовой, текстовой, графической, звуковой информации в цифровую необходимо применить кодирование. Кодирование – это преобразование данных одного типа через данные другого типа. В ЭВМ применяется система двоичного кодирования, основанная на представлении данных последовательностью двух знаков: 1 и 0, которые называются двоичными цифрами (binary digit – сокращенно bit).

Восемь последовательных бит составляют байт. В одном байте можно закодировать значение одного символа из 256 возможных ($256 = 2$ в степени 8). Более крупной единицей информации является килобайт (Кбайт), равный 1024 байтам ($1024 = 2$ в степени 10). Еще более крупные единицы измерения данных: мегабайт, гигабайт, терабайт ($1 \text{ Мбайт} = 1024 \text{ Кбайт}; 1 \text{ Гбайт} = 1024 \text{ Мбайт}; 1 \text{ Тбайт} = 1024 \text{ Гбайт}$).

Целые числа кодируются двоичным кодом путем деления числа на два. Для кодирования нечисловой информации все возможные значения кодируемой информации нумеруются и эти номера кодируются с помощью двоичного кода. Для представления текстовой информации используется таблица нумерации символов или таблица кодировки символов, в которой каждому символу соответствует целое число (порядковый номер). Восемь двоичных разрядов могут закодировать 256 различных символов.

Система счисления – это способ записи чисел с помощью заданного набора специальных знаков (цифр).

Существуют системы позиционные и непозиционные. (есть еще смешанные, пример – денежные знаки - чтобы получить некоторую сумму в рублях, нужно использовать некоторое количество денежных знаков различного достоинства)

В непозиционных системах счисления вес цифры не зависит от позиции, которую она занимает в числе. Так, например, в римской системе счисления в числе XXXII (тридцать два) вес цифры X в любой позиции равен просто десяти.

В позиционных системах счисления вес каждой цифры изменяется в зависимости от ее позиции в последовательности цифр, изображающих число.

Любая позиционная система характеризуется своим основанием. Основание позиционной системы счисления – это количество различных знаков или символов, используемых для изображения цифр в данной системе (любое натуральное число).

Десятичная система счисления

В этой системе 10 цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, однако информацию несет не только цифра, но и место, на котором цифра стоит (то есть ее позиция). Особую роль играют число 10 и его степени: 10, 100, 1000 и т. д. Самая правая цифра числа показывает число единиц, вторая справа – число десятков, следующая – число сотен и т. д.

Двоичная система счисления

В этой системе всего две цифры – 0 и 1. Особую роль здесь играет число 2 и его степени: 2, 4, 8 и т. д. Самая правая цифра числа показывает число единиц, следующая цифра – число двоек, следующая – число четверок и т. д. Двоичная система счисления позволяет закодировать любое натуральное число – представить его в виде последовательности нулей и единиц.

Восьмеричная система счисления

В этой системе счисления 8 цифр: 0, 1, 2, 3, 4, 5, 6, 7. Цифра 1, указанная в самом младшем разряде, означает, как и в десятичном числе, просто единицу. Та же цифра 1 в следующем разряде означает 8, в следующем – 64 и т. д. Число 100 (восьмеричное) есть не что иное, как 64 (десятичное). Чтобы перевести в двоичную систему, например, число 611 (восьмеричное), надо заменить каждую цифру эквивалентной ей двоичной триадой (тройкой цифр).

Шестнадцатеричная система счисления

В качестве первых 10 из 16 шестнадцатеричных цифр взяты привычные цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, а вот в качестве остальных 6 цифр используют первые буквы латинского алфавита: A, B, C, D, E, F. Цифра 1, записанная в самом младшем разряде, означает просто единицу. Та же цифра 1 в следующем – 16 (десятичное), в следующем – 256 (десятичное) и т. д. Цифра F, указанная в самом младшем разряде, означает 15 (десятичное). Перевод из шестнадцатеричной системы в двоичную и обратно производится аналогично тому, как это делается для восьмеричной системы.

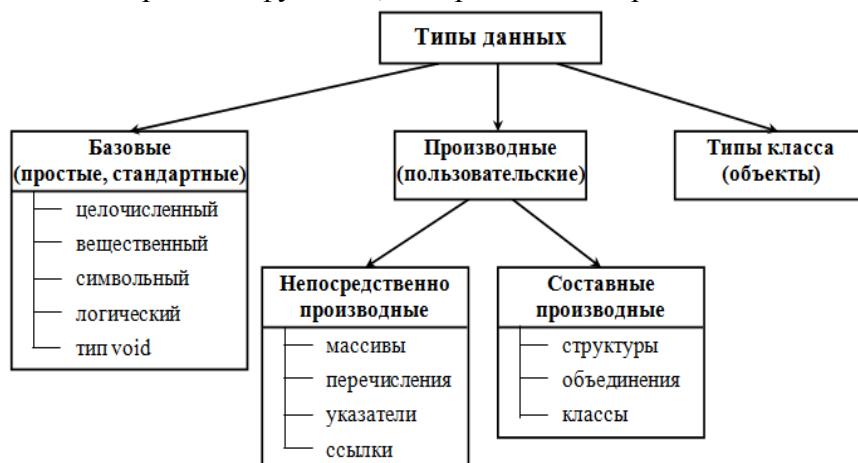
(связью является перевод из одной системы счисления в другую)

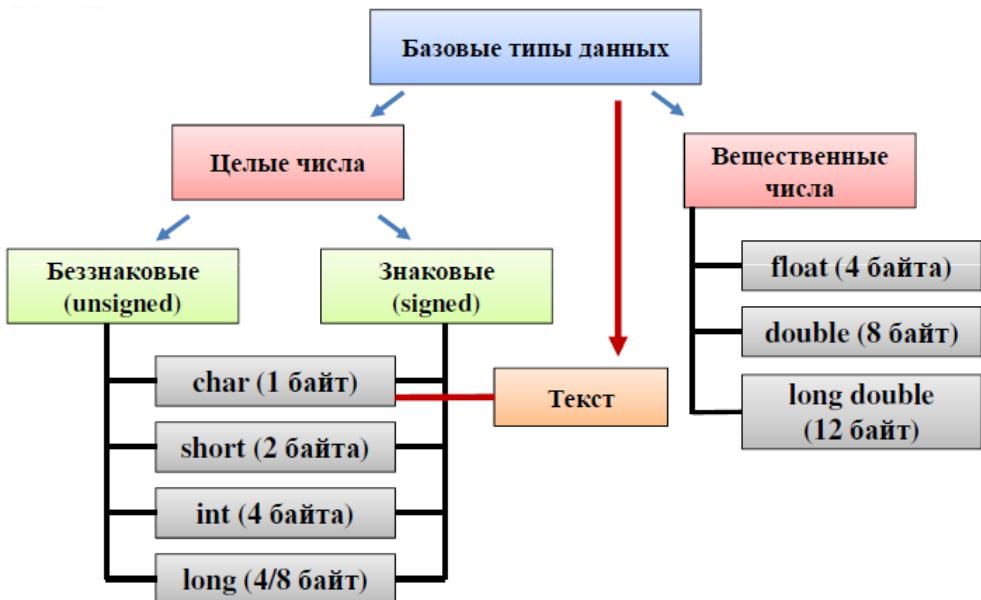
3. Типы данных. Внутреннее представление базовых типов данных. Преобразование типов данных

Тип данных – это множество допустимых значений, которые может принимать тот или иной объект, а также множество допустимых операций, которые применимы к нему. В современном понимании тип также зависит от внутреннего представления информации.

Таким образом, данные различных типов хранятся и обрабатываются по-разному. Тип данных определяет:

- внутреннее представление данных в памяти компьютера;
- объем памяти, выделяемый под данные;
- множество (диапазон) значений, которые могут принимать величины этого типа;
- операции и функции, которые можно применять к данным этого типа.





Преобразования типов выполняются, если операнды, входящие в выражения, имеют различные типы. Ниже приведена последовательность преобразований.

Любые операнды типа char, unsigned char или short преобразуются к типу int по правилам:

- char расширяется нулем или знаком в зависимости от умолчания для char;
- unsigned char расширяется нулем;
- signed char расширяется знаком;
- short, unsigned short и enum при преобразовании не изменяются.
- Затем любые два операнда становятся либо int, либо float, double или long double.

Преобразование других типов данных осуществляется следующим образом:

1. Если один из operandов имеет тип long double, то другой преобразуется к типу long double.
2. Если один из operandов имеет тип double, то другой преобразуется к типу double.
3. Если один из operandов имеет тип float, то другой преобразуется к типу float.
4. Иначе, если один из operandов имеет тип unsigned long, то другой преобразуется к типу unsigned long.
5. Иначе, если один из operandов имеет тип long, то другой преобразуется к типу long.
6. Иначе, если один из operandов имеет тип unsigned, то другой преобразуется к типу unsigned.
7. Иначе оба операнда должны иметь тип int.

Тип результата тот же, что и тип участвующих в выражении operandов.

Представление отрицательного числа в дополнительном коде (Внутреннее представление базовых типов данных):

При записи числа в дополнительном коде старший разряд является знаковым. Если его значение равно 0, то в остальных разрядах записано положительное двоичное число, совпадающее с прямым кодом.

Двоичное 8-разрядное число со знаком в дополнительном коде может представлять любое целое в диапазоне от -128 до +127. Если старший разряд равен нулю, то наибольшее целое число, которое может быть записано в оставшихся 7 разрядах, равно $2^7 - 1 = 127$.

4. Указатели в языке Си. Тип указателя. Действия над указателями.

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Каждая переменная в памяти имеет свой адрес — номер первой ячейки, где она расположена, а также свое значение. Указатель — это тоже переменная, которая размещается в памяти. Она тоже имеет адрес, а ее значение является адресом некоторой другой переменной. Переменная, объявленная как указатель, занимает 4 байта в оперативной памяти (в случае 32-битной версии компилятора).

Указатель, как и любая переменная, должен быть объявлен.

Общая форма объявления указателя

тип *ИмяОбъекта;

Тип указателя — это тип переменной, адрес которой он содержит.

Для работы с указателями в Си определены две операции:

- операция * (звездочка) — позволяет получить значение объекта по его адресу — определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;
- операция & (амперсанд) — позволяет определить адрес переменной.

Например,

```
char c; //  
char *p; //  
p = &c; // p = адрес c
```

переменная
указатель

5. Алгоритмы на базе циклических конструкций. Элементарная теория чисел.

Цикл — это такая форма организации действий, при которой одна последовательность действий повторяется несколько раз (или ни разу), до тех пор, пока выполняются некоторые условия. В алгоритмические структуры цикл входит серия команд, выполняемая многократно. Такая последовательность команд называется телом цикла.

Циклические алгоритмические структуры бывают двух видов:

- циклы со счетчиком (с заранее известным числом повторений);
- циклы с условием (с заранее неизвестным числом повторений);

Циклической конструкцией со счетчиком является цикл “For или Для”, его используют когда заранее известно, какое число повторений тела цикла необходимо выполнить.

Цикл “While или Пока”. Цикл “While” является циклической конструкцией с условием, т.е. это такой цикл, где тело цикла выполняется до тех пор, пока выполняются некоторые условия.

Существует также пустой цикл - это цикл без тела цикла. В большинстве случаев он применяется для создания пауз в программах.

Наиболее часто в алгоритмах и программах применяются два вида циклов. Это циклы “While или Пока” и “For или Для”.

В основе многих циклических конструкций (или их фрагментов) лежит фундаментальная идея о том, что результат вычислений на каждом шаге цикла должен зависеть от результатов предыдущего шага.

```

#include <stdio.h>
int main()
{
    int i, S, N, a;
    scanf("%d", &N);
    i = 1; S = 0;
    while( i <= N ){
        scanf("%d", &a);
        S = S + a;
        i = i + 1;
    }
    printf("Sum a[1...%d] = %d\n", N, S);
    return 0;
}

```

Новое значение S выражается через предыдущее.
То же самое относится к i . И, в принципе, к a !

В условии наличия циклов программа как бы разделена на дискретные промежутки, на которых значения переменных неизменны, изменение любой из переменных означает изменение состояния всей программы.

В элементарной теории чисел целые числа изучаются без использования методов других разделов математики. Такие вопросы, как [делительность целых чисел](#), [алгоритм Евклида](#) для вычисления [наибольшего общего делителя](#) и [наименьшего общего кратного](#), разложение числа на [простые множители](#), теория сравнений, диофантовы уравнения, построение [магических квадратов](#), [совершенные числа](#), [числа Фибоначчи](#), [малая теорема Ферма](#), [теорема Эйлера](#), [задача о четырёх кубах](#), относятся к этому разделу.

Наибольший общий делитель (НОД) для двух целых чисел m и n называется наибольший из их общих делителей.

НОД существует и однозначно определён, если хотя бы одно из чисел m или n не ноль. Одним из способов нахождения (n,m) и $[n,m]$ является разложение чисел n и m на простые множители.

Пусть

$$n = p_1^{d_1} \cdot p_2^{d_2} \cdot p_3^{d_3} \cdots p_k^{d_k}$$

$$m = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3} \cdots p_k^{e_k}$$

Тогда

$$(n, m) = p_1^{\min(d_1, e_1)} \cdot p_2^{\min(d_2, e_2)} \cdot p_3^{\min(d_3, e_3)} \cdots p_k^{\min(d_k, e_k)}$$

$$[n, m] = p_1^{\max(d_1, e_1)} \cdot p_2^{\max(d_2, e_2)} \cdot p_3^{\max(d_3, e_3)} \cdots p_k^{\max(d_k, e_k)}$$

Примеры:

$$60 = 2 \cdot 2 \cdot 3 \cdot 5 = 2^2 \cdot 3 \cdot 5 = 2^2 \cdot 3^1 \cdot 5^1 \cdot 7^0$$

$$84 = 2 \cdot 2 \cdot 3 \cdot 7 = 2^2 \cdot 3 \cdot 7 = 2^2 \cdot 3^1 \cdot 5^0 \cdot 7^1$$

$$(60, 84) = 2^2 \cdot 3^1 \cdot 5^0 \cdot 7^0 = 12$$

$$[60, 84] = 2^2 \cdot 3^1 \cdot 5^1 \cdot 7^1 = 420$$

$$17\ 640 = 2^3 \cdot 3^2 \cdot 5 \cdot 7^2 = 2^3 \cdot 3^2 \cdot 5 \cdot 7^2 \cdot 11^0 \cdot 17^0$$

$$26\ 180 = 2^2 \cdot 5 \cdot 7 \cdot 11 \cdot 17 = 2^3 \cdot 3^0 \cdot 5 \cdot 7^1 \cdot 11^1 \cdot 17^1$$

$$(17\ 640, 26180) = 2^2 \cdot 3^0 \cdot 5 \cdot 7^1 \cdot 11^0 \cdot 17^0 = 140$$

$$[17\ 640, 26180] = 2^3 \cdot 3^2 \cdot 5 \cdot 7^2 \cdot 11^1 \cdot 17^1 = 3\ 298\ 680$$

Линейный метод поиска НОД

Пусть даны целые числа n и m . Необходимо найти их НОД (n, m).

Решение:

По определению (n, m) – это наибольшее целое число, на которое нацело делится и n и m . Поэтому наиболее очевидным способом вычисления (n, m) является перебор ВСЕХ чисел x , которые могут быть делителями и n и m .

Очевидно, что некоторое целое число n не может поделиться нацело на число x : $x > n$. Поэтому естественным будет искать делители в диапазоне $[2, \min(n, m)]$:



```
input n, m
if m > n then
    t ← n, n ← m, m ← t
if (n mod m = 0) then
    g ← m
else
    g ← [m/2]
    while (m mod g) ≠ 0 or (n mod g) ≠ 0 do
        g ← g - 1
output g
```

Наименьшее общее кратное (НОК) двух целых чисел m и n есть наименьшее натуральное число, которое делится как на m , так и на n .

```
ввод n, m
if m > n then
    t ← n, n ← m, m ← t
    i ← 1, e ← m, f ← 0
    while (i < e и f ≠ 1) do
        t1 ← m div i
        t2 ← m mod i
        if (t2 = 0 и (n mod t1 = 0)) then
            g ← t1, f ← 0
        else if (t2 = 0 и (n mod i = 0)) then
            g ← i
        i ← i + 1, e ← t1
вывод g
```

6. Понятие массива в языках программирования, области его применения.

Массивы содержат набор данных одинакового типа. Этот тип может быть простым типом, структурой или классом. Члены массива называются элементами. К элементам можно получить доступ, используя индекс. Элементы могут быть инициализированы определенным значением при определении массива.

```
int arr[10]; // массив десяти целых чисел
```

В отличие от векторов, массивы имеют фиксированный размер. Массив может иметь несколько размерностей (одномерный, двумерный, трехмерный). Массив называется одномерным, если для доступа к его элементам достаточно одного индекса. Двумерный массив – это массив массивов. Адрес массива может быть использован как аргумент функции; сам массив при этом не копируется. Массив может быть использован как переменная класса. Необходимо заботиться о том, чтобы данные не были помещены за пределы массива.

Нельзя инициализировать массив как копию другого массива, недопустимо также присвоение одного массива другому.

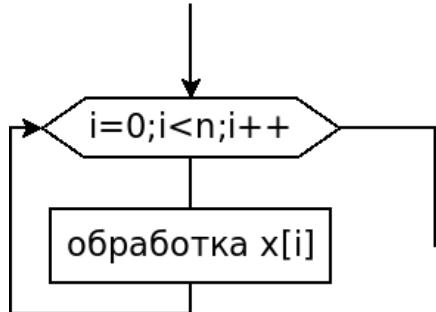
Поскольку размеры массивов постоянны, они иногда обеспечивают лучшую производительность во время выполнения приложений. Но это преимущество приобретается за счет потери гибкости.

Областями применения массивов являются:

- Матричная алгебра, экстраполяция, интерполяция;
- Представление других структур: графов, деревьев;
- Управляющие и информационные таблицы в операционных системах, трансляторах, системами управления базами данных (СУБД);
- Числовые массивы в вычислительных задачах;
- Таблицы – массивы с элементами типа «запись».

7. Алгоритмы обработки массивов. Задача поиска простых чисел. Линейный алгоритм. Алгоритм Эратосфена.

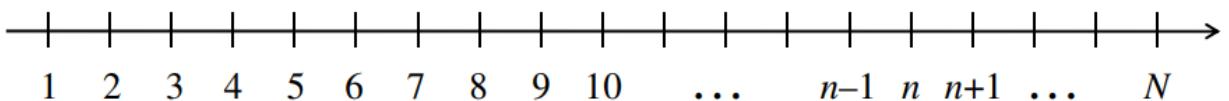
Все манипуляции с массивами в C++ осуществляются поэлементно. Организовывается цикл, в котором происходит последовательное обращение к нулевому, первому, второму и т.д. элементам массива. В общем виде алгоритм обработки массива выглядит так, как показано на рисунке.



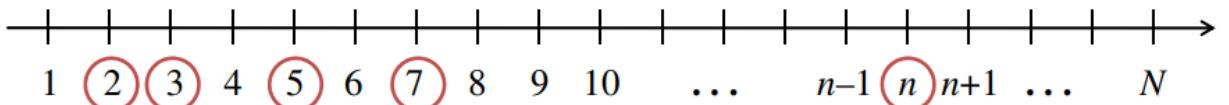
Алгоритмы, с помощью которых обрабатывают одномерные массивы, похожи на обработку последовательностей (вычисление суммы, произведения, поиск элементов по определённому признаку, выборки и т. д.). Отличие заключается в том, что в массиве одновременно доступны все его компоненты, поэтому становится возможной, например, сортировка его элементов и другие, более сложные преобразования.

Простое число – это натуральное число, имеющее ровно два различных натуральных делителя: единицу и само себя.

Задача поиска простых чисел в заданном диапазоне.



Дан диапазон натуральных чисел $[2, N]$. Требуется выбрать из него только те числа, которые являются простыми:



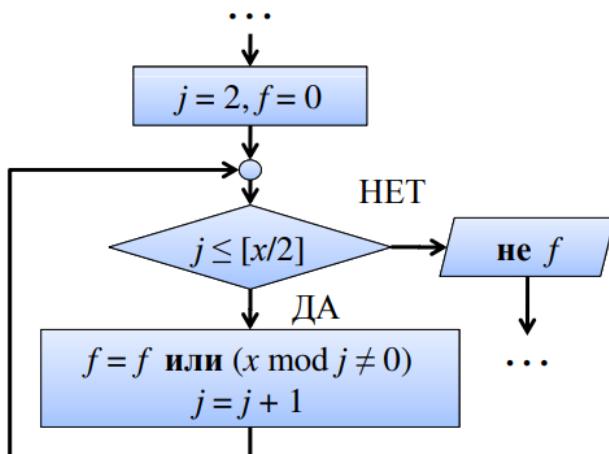
Для того, чтобы проверить, является ли число x простым достаточно проверить, делится ли оно на одно из чисел диапазона $[2, x/2]$. Числа, прошедшие проверку на простоту могут быть помещены в специально отведенный для этого массив, размерность которого совпадает с исходным (пессимистическое выделение памяти).

x – целое число, которое проверяется на простоту.

На каждой итерации проверяется потенциальный делитель

$$j \in [2, [x/2]].$$

Если $x \bmod j = 0$ (делится нацело), то переменная-флаг $f = 1$.



На выходе значение флага f определяет результат:

- если $f = 0$ ((не f) = 1), то среди чисел диапазона $[2, [x/2]]$ не нашлось ни одного делителя – число простое.
- иначе $f = 1$ ((не f) = 0) – число составное.

$x \leftarrow 2, n \leftarrow 0 // x$ – проверяемые числа, n – счетчик простых чисел

while $x \leq N$ **do**

// j – потенциальные делители, f – флаг, 0 => простое число

$j \leftarrow 2, f \leftarrow 0$

while $j \leq (x \bmod 2)$ **и** $f \neq 1$ **do**

if $(x \bmod j = 0)$ **then** // если x делится на j – x не простое!

$f \leftarrow 1$

$j \leftarrow j + 1$ // перейти к следующему делителю

if $f = 0$ **then** // если флаг $f = 0$, то x – простое число

$n \leftarrow n + 1$ // счетчик n – первый свободный элемент

$primes[n] \leftarrow x$

$x \leftarrow x + 1$

Решето Эратосфена — алгоритм нахождения всех простых чисел до некоторого целого числа N , который приписывают древнегреческому математику Эратосфену Киренскому.

1. Выписать подряд все целые числа от двух до N ($2, 3, 4, \dots, N$).
2. Пусть переменная p изначально равна двум – первому простому числу.
3. Считая от p шагами по p , зачеркнуть в списке все числа от $2p$ до n кратные p (то есть числа $2p, 3p, 4p, \dots$).
4. Найти первое незачеркнутое число в списке, большее чем p , и присвоить значению переменной p это число.
5. Повторять шаги 3 и 4, пока незачеркнутое число есть.

1. Выписать подряд все целые числа от двух до N ($2, 3, 4, \dots, N$).

$2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad \dots \quad N$

2. Пусть переменная $p = 2$ (первому простому числу).



3. Считая от p шагами по p , зачеркнуть в списке все числа от $2p$ до n кратные p (то есть числа $2p, 3p, 4p, \dots$).



4. Найти первое незачеркнутое число в списке, большее чем p , и присвоить значению переменной p это число: $p = 3$.

5. Повторять шаги 3 и 4, пока незачеркнутое число есть.

1. Выписать подряд все целые числа от двух до N ($2, 3, 4, \dots, N$).

$2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad \dots \quad N$

2. $p = 3$



3. Считая от p шагами по p , зачеркнуть в списке все числа от $2p$ до n кратные p (то есть числа $2p, 3p, 4p, \dots$).



4. Найти первое незачеркнутое число в списке, большее чем p , и присвоить значению переменной p это число: $p = 5$.

5. Повторять шаги 3 и 4, пока незачеркнутое число есть.

Псевдокод алгоритма Эратосфена

```
for i ← 1 to (N – 1) do // Заполнить массив числами диапазона
    primes[i] ← i + 1
    i ← 1
    while i < N do // основной цикл
        s ← primes[i]
        p ← i + s
        while p < N do // Вычеркивание (обнуление) не простых чисел
            primes[p] ← 0
            p ← p + s
        i ← i + 1
        while i < N и primes[i] = 0 do // Первое невычеркнутое
            i ← i + 1
```

8. Понятие сортировки. Алгоритмы сортировки. Оценка алгоритмов сортировки.

Задача сортировки (sorting problem)

Дано: последовательность из n чисел $\langle a_1, a_2, a_3, \dots, a_n \rangle$

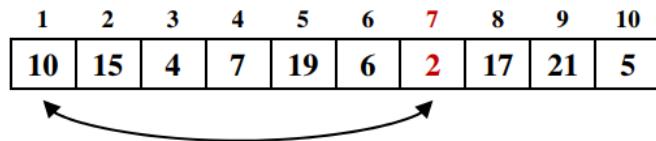
Необходимо: переставить элементы последовательности так, чтобы для любых элементов новой последовательности $\langle a'_1, a'_2, a'_3, \dots, a'_n \rangle$ выполнялось соотношение:

$a'_1 \leq a'_2 \leq a'_3 \leq \dots \leq a'_n$ (сортировка по возрастанию)

- **Внутренние методы сортировки (Internal sort)** – сортируемые элементы полностью размещены в оперативной памяти компьютера
- **Внешняя сортировка (External sort)** – элементы размещены на внешней памяти (жесткий диск, USB-флеш)
- Алгоритм сортировки не использующий дополнительной памяти (кроме сортируемого массива) называется **алгоритмом сортировки на месте (in-place sort)**

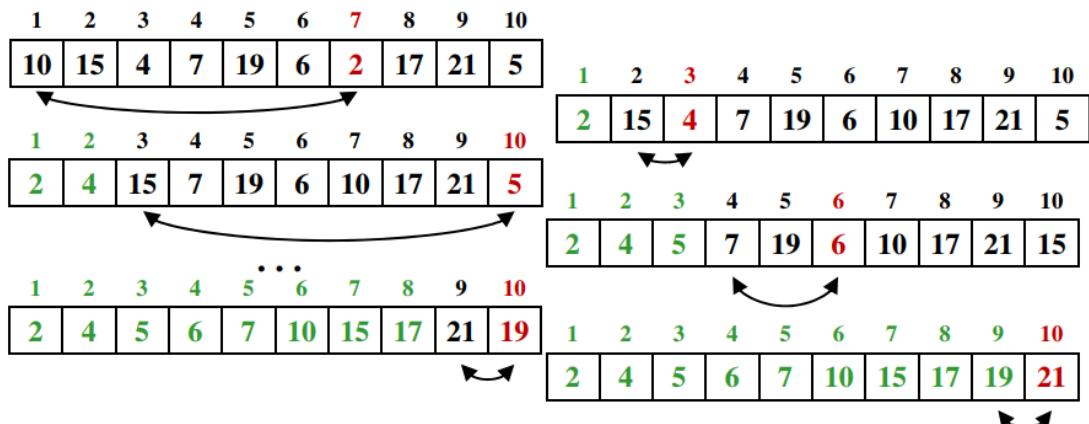
Алгоритм сортировки выбором

Суть алгоритма заключается в последовательном формировании упорядоченной последовательности слева направо. На каждом шаге рассматривается фрагмент массива с i по n -й элементы. Среди них выбирается наименьший, который занимает первое место диапазона (на место i -го элемента). При этом i -й элемент перемещается на позицию, в которой найден минимум:



После этого считается, что элементы массива с 1 по i отсортированы. Поэтому далее процедура повторяется для диапазона элементов с $i+1$ по n .

После каждого прохода алгоритма сортировки выбором размер отсортированной части увеличивается на 1 элемент:



Учитывая, что на последнем проходе рассматривается один элемент, количество проходов для получения полностью отсортированной последовательности составляет $n - 1$.

```

 $i \leftarrow 1$ 
while  $i < n$  do
     $m \leftarrow i, k \leftarrow i + 1$ 
    while  $k \leq n$  do
        if  $a[k] < a[m]$  then
             $m \leftarrow k$ 
         $k \leftarrow k + 1$ 
         $t \leftarrow a[i], a[i] \leftarrow a[m], a[m] \leftarrow t$ 
     $i \leftarrow i + 1$ 

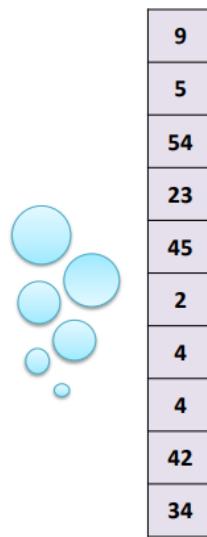
```

Сортировка пузырьком (Bubble sort) — для каждой пары индексов производится обмен, если элементы расположены не по порядку.

```

function BubbleSort( $v[0:n - 1]$ ,  $n$ )
    swapped = True
    while swapped do
        swapped = False
        for  $i = 1$  to  $n - 1$  do
            if  $v[i - 1] > v[i]$  then
                swap( $v[i - 1]$ ,  $v[i]$ )
                swapped = True
            end if
        end for
    end while
end function

```

 $T_{\text{BubbleSort}} = O(n^2)$


“Легкие” элементы перемещаются
(всплывают) в начало массива

Сортировка вставками (Insertion sort) — Определяем, где текущий элемент должен находиться в упорядоченном списке, и вставляем его туда.

```

function InsertionSort(A[1:n], n)
    for i = 2 to n do
        key = A[i]
        /* Вставляем A[i] в упорядоченный подмассив A[1..i-1] */
        j = i - 1
        while j > 0 and A[j] > key do
            A[j + 1] = A[j]
            j = j - 1
        end while
        A[j + 1] = key
    end for
end function

```

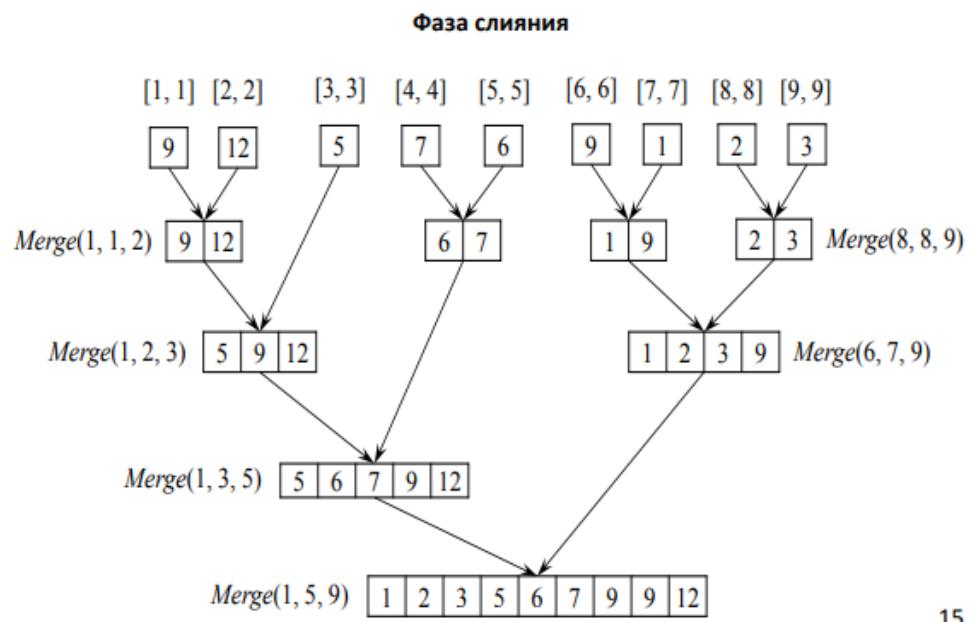
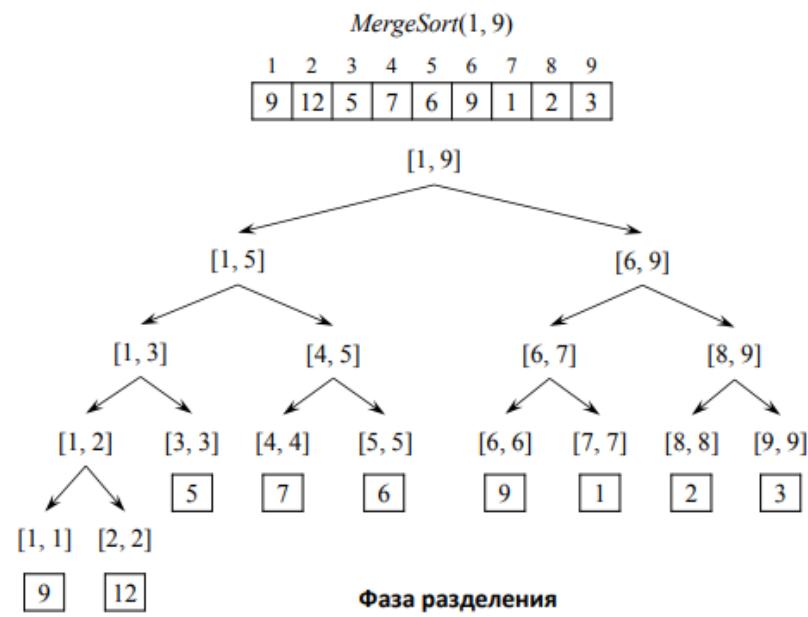
- Двигаемся по массиву слева направо:
от 2-го до n -го элемента
- На шаге i имеем упорядоченный подмассив $A[1..i-1]$
и элемент $A[i]$, который необходимо вставить в этот
подмассив

- В **худшем** случае цикл while всегда доходит до первого элемента массива – на вход поступил массив, упорядоченный по убыванию.
- Для вставки элемента $A[i]$ на своё место требуется $i - 1$ итерация цикла while; на каждой итерации выполняем с действий;
- Учитывая, что нам необходимо найти позиции для $n - 1$ элемента, время $T(n)$ выполнения алгоритма в худшем случае равно $T(n)=O(n^2)$

Лучший случай для Insertion sort?

- Массив уже упорядочен
- Алгоритм сортировки вставкой является устойчивым – не меняет относительный порядок следования одинаковых ключей
- Использует константное число дополнительных ячеек памяти (переменные i , key и j), что относит его к классу алгоритмов сортировки на месте (in-place sort).
- Кроме того, алгоритм относится к классу online- алгоритмов – он обеспечивает возможность упорядочивания массивов при динамическом поступлении новых элементов

Сортировка слиянием (Merge sort) — рекурсивный алгоритм сортировки сравнением, основанный на методе декомпозиции (decomposition). Выстраиваем первую и вторую половину списка отдельно, а затем объединяем упорядоченные списки.



```

function MergeSort(A[1:n], low, high)
    if low < high then
        mid = floor((low + high) / 2)
        MergeSort(A, low, mid)
        MergeSort(A, mid + 1, high)
        Merge(A, low, mid, high)
    end if
end function

```

- Сортируемый массив $A[low..high]$ *разделяется* (partition) на две максимально равные по длине части
- Первая часть содержит $\lceil n/2 \rceil$ элементов, вторая – $\lfloor n/2 \rfloor$ элементов
- Подмассивы рекурсивно сортируются

```

function Merge(A[1:n], low, mid, high)
    for i = low to high do
        B[i] = A[i] /* Создаем копию массива A */
    end for

    l = low      /* Номер первого элемента левого подмассива */
    r = mid + 1 /* Номер первого элемента правого подмассива */
    i = low
    while l <= mid and r <= high do
        if B[l] <= B[r] then
            A[i] = B[l]
            l = l + 1
        else
            A[i] = B[r]
            r = r + 1
        end if
        i = i + 1
    end while

    while l <= mid do
        /* Копируем оставшиеся элементы из левого подмассива */
        A[i] = B[l]
        l = l + 1
        i = i + 1
    end while
    while r <= high do
        /* Копируем оставшиеся элементы из правого подмассива */
        A[i] = B[r]
        r = r + 1
        i = i + 1
    end while
end function

```

- Функция Merge требует порядка $\Theta(n)$ ячеек памяти для хранения копии B сортируемого массива.
- Сравнение и перенос элементов из массива B в массив A требует $\Theta(n)$.

- Время $T(n)$ работы алгоритма включает время сортировки левого подмассивов длины $n/2$ и правого – с числом элементов $n/2$, а также время $\Theta(n)$ слияния подмассивов после их рекурсивного упорядочивания

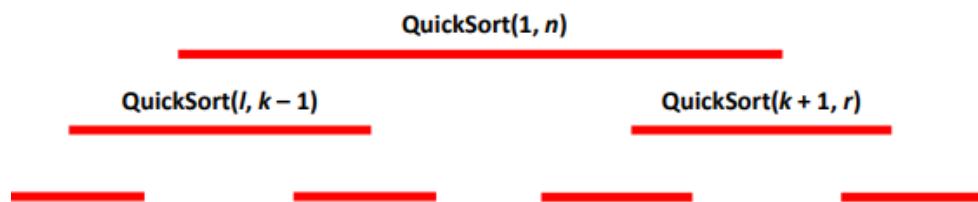
$$T(n) = T([n/2]) + T([n/2]) + \Theta(n)$$

- Необходимо решить это рекуррентное уравнение – получить выражение для $T(n)$ без рекуррентности.

Быстрая сортировка (Quicksort) - широко известен как быстрейший из известных для упорядочения больших случайных списков; с разбиением исходного набора данных на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй половины; затем алгоритм применяется рекурсивно к каждой половине.

1. Из элементов $v[1], v[2], \dots, v[n]$ выбирается **опорный элемент** (pivot element)
 - Опорный элемент желательно выбирать так, чтобы его значение было близко к среднему значению всех элементов
 - Вопрос о выборе опорного элемента открыт (первый/последний, средний из трех, случайный и т.д.)
2. Массив разбивается на 2 части: элементы массива переставляются так, чтобы элементы расположенные левее опорного были не больше (\leq), а расположенные правее – не меньше него (\geq). На этом шаге определяется граница разбиения массива.
3. Шаги 1 и 2 рекурсивно повторяются для левой и правой частей

```
function QuickSort(v[1:n], n, l, r)
  if l < r then
    k = Partition(v, n, l, r)      // Разбиваем
    QuickSort(v, n, l, k - 1)      // Левая часть
    QuickSort(v, n, k + 1, r)      // Правая часть
  end if
end function
```



```

function Partition( $v[1:n]$ ,  $n$ ,  $l$ ,  $r$ )
    pivot_idx =  $r$           /* Выбрали индекс опорного элемента */
    swap( $v[pivot\_idx]$ ,  $v[r]$ )
    pivot =  $v[r]$ 
     $i = l - 1$ 
    for  $j = l$  to  $r - 1$  do
        if  $v[j] \leq pivot$  then
             $i = i + 1$ 
            swap( $v[i]$ ,  $v[j]$ )
        end if
    end for
    swap( $v[i + 1]$ ,  $v[r]$ )
    return  $i + 1$ 
end function

```

2	8	7	1	3	5	6	4
$i = 1$							$r = n$
$i = 0$							$pivot = 4$
$j = 1:$ swap($v[1], v[1]$)							
2	1	7	8	3	5	6	4
$j = 4:$ swap($v[2], v[4]$)							
2	1	3	8	7	5	6	4
$j = 5:$ swap($v[3], v[5]$)							
2	1	3	8	7	5	6	4
$swap(v[4], v[8])$							
2	1	3	4	7	5	6	8
$return 4$							
25							

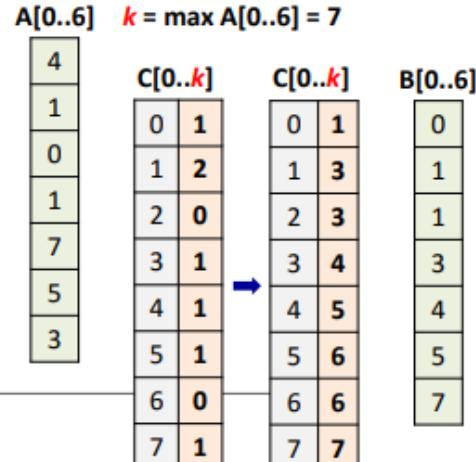
Сортировка подсчетом (counting sort)

Алгоритм 3.4. Сортировка подсчетом

```

1 function COUNTINGSORT( $A[0..n - 1]$ ,  $B[0..n - 1]$ ,  $k$ )
2   for  $i = 0$  to  $k$  do
3      $C[i] = 0$ 
4   end for
5   for  $i = 0$  to  $n - 1$  do
6      $C[A[i]] = C[A[i]] + 1$ 
7   end for
8   for  $i = 1$  to  $k$  do
9      $C[i] = C[i] + C[i - 1]$ 
10  end for
11  for  $i = n - 1$  to  $0$  do
12     $C[A[i]] = C[A[i]] - 1$ 
13     $B[C[A[i]]] = A[i]$ 
14  end for
15 end function

```



- Не использует операцию сравнения!
- Целочисленная сортировка (integer sort)
- Вычислительная сложность $O(n + k)$
- Сложность по памяти $O(n + k)$

Алгоритм	Лучший случай	Средний случай	Худший случай	Память	Свойства
Сортировка вставками (Insertion Sort)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Устойчивый, на месте, online
Сортировка выбором (Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Устойчивость зависит от реализации, на месте
Быстрая сортировка (Quick Sort)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Неустойчивая
Сортировка слиянием (Merge sort)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Устойчивая

13. Многофайловые программы. Классы памяти переменных.

При разработке достаточно больших программ бывает удобным разрабатывать программу не в виде одного файла, а в виде нескольких. В отдельном файле сохраняем функцию **main()**, подпрограммы — каждую в отдельном файле или группируем по назначению.

Что это даёт? Подпрограмма, сохранённая в отдельном файле, может быть очень легко использована в другой программе. Достаточно будет только подключить к проекту новой программы файл с этой подпрограммой.

А теперь представим, что программа у нас несколько больше и содержит десяток файлов исходного кода. Файл aa.c требует функций из файла bb.c, dd.c, ee.c. В свою очередь dd.c вызывает функции из ee.c и ff.c, а эти два последних файла активно пользуются неким файлом stars.c и одной из функций в bb.c. Программист замучится сверять, что чего вызывает откуда и куда, где и какие объявления надо прописывать. Поэтому все прототипы (объявления) функций проекта, а также совместно используемые символические константы и макросы выносят в отдельный файл, который подключают к каждому файлу исходного кода. Такие файлы называются заголовочными. В отличие от заголовочных файлов стандартной библиотеки, заголовочные файлы, которые относятся только к вашему проекту, при подключении к файлу исходного кода заключаются в кавычки, а не скобки.

Итак, более грамотно будет не добавлять объявления функций в файл main.c, а создать заголовочный файл, например, myprint.h и поместить туда прототипы функций l2r() и r2l(). А в файле main.c следует прописать директиву препроцессора: #include "myprint.h"

Любой заголовочный файл C/C++ должен иметь следующую структуру.

```
#ifndef ИМЯ_ЗАГОЛОВОЧНОГО_ФАЙЛА
#define ИМЯ_ЗАГОЛОВОЧНОГО_ФАЙЛА

/* здесь помещается остальной текст заголовочного файла */

#endif
```

Директивы ifndef-define-endif, которые обрамляют любой грамотно оформленный заголовочный файл, обеспечивают то, что любой заголовочный файл будет включён в любой исходный файл не более одного раза.

Более подробно. Каждому заг. файлу «вручную» ставится в соответствие некоторый «символ», обычно связанный с именем этого файла, чтобы обеспечить уникальность. В первой строке проверяется, был ли уже определён этот символ ранее, если да, то весь остальной текст игнорируется. Если нет, то этот символ определяется, а затем вставляется и весь остальной текст заголовочного файла. Последняя строка (endif) просто означает закрытие такого «условного оператора».

Класс памяти переменной (англ. *Storage class*) — определяет область видимости переменной, а также как долго переменная находится в памяти.

Переменная в С/С++ должна принадлежать ровно одному классу памяти, что указывается с помощью ключевого слова, который пишется перед типом переменной.

- `auto` — автоматическая (локальная). Автоматические переменные создаются при входе в функцию и уничтожаются при выходе из неё. Они видны только внутри функции или блока, в которых определены.
- `static` — статическая переменная (локальная). 1) Если `static` — внутри функции. Для таких переменных область видимости обычна (внутри функции), но время жизни постоянное (значение сохраняется между вызовами функции). 2) `static` вне функции имеет другое значение.
- `extern` — внешняя (глобальная) переменная. Внешние переменные доступны везде, где описаны, а не только там, где определены. Использование ключевого слова `extern` позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле. Для таких переменных связывание с адресом происходит на этапе компоновки.
- `register` — регистровая переменная (локальная). Это слово является всего лишь «рекомендацией» компилятору помещать часто используемую переменную в регистры процессора для ускорения программы.

Класс памяти можно не указывать, тогда действуют следующие умолчания:

- переменные, описанные внутри функции или блока, считаются локальными (`auto`)
- переменные, описанные вне всех функций, считаются внешними.
- функции считаются внешними.

Статическая переменная, описанная вне любой функции, становится внешней статической. Разница между внешней переменной и внешней статической переменной заключается в области их действия. Обычная внешняя переменная может использоваться функциями в любом файле, а внешняя статическая переменная может использоваться только функциями того же самого файла, причем после определения переменной.

14. Структурный тип данных. Тип данных объединение.

Примером структуры может послужить любой объект, для которого описывается ряд его характеристик, имеющих значение в данной программе. Например, для книг это может быть название, автор, количество страниц; для окружности — координаты центра, диаметр, цвет. На языке программирования С объявление вышеупомянутых структурных типов данных может выглядеть так:

```
struct book {  
    char title[50];  
    char author[30];  
    int pages;  
};  
struct circle {  
    int x, y;  
    float dia;  
    char color[10];  
};
```

В данном случае мы как бы создаем новый тип данных, но еще не объявляем переменных этих типов. Обратите внимание на точку с запятой в конце объявлений.

Чаще переменные структур объявляются так:

```
struct circle a, b, c;  
struct book mybook;
```

Здесь объявляются три структуры типа circle и одна структура типа book. Можно объявлять типы структур и их переменные по-иному, но мы для избежания путаницы рассматривать другие способы не будем.

Каждая переменная типа circle содержит четыре элемента (или поля) — x, y, dia, color. Можно сказать, что они представляют собой вложенные переменные. Причем эти переменные разных типов. Таким образом переменная-структура позволяет объединить под одним именем ряд разнородных данных. Обычно это нужно для удобства обработки данных. Если нельзя было бы создавать структуры, то пришлось бы создавать множество независимых переменных или ряд массивов, явной взаимосвязи между которыми не было бы. Структуру же позволяют объединять взаимосвязанные данные.

Объявив переменную структурного типа, мы можем получить доступ к каждому ее элементу для присваивания, изменения или получения значения:

```
a.x = 10; a.dia = 2.35;  
printf("% .2f ", a.dia);
```

Значение элементов структуры можно сразу определять при объявлении переменной, что похоже на инициализацию массивов:

```
struct book lang_c = {"Language C", "Ritchi", 99};
```

Значение переменной-структуры можно присвоить переменной того же типа:

```
struct book { char *title, *author; int pages; };  
struct book old, new;  
old.title = "GNU/Linux"; old.author = "people"; old.pages = 20213;  
new = old;  
new.pages += 2000;  
printf("%d, %d\n", old.pages, new.pages);
```

В четвертой строке кода данные переменной old присваиваются new. В итоге вторая структура содержит копию данных первой.

Структуры и функции

Структуры-переменные можно передавать в функции в качестве параметров и возвращать их оттуда. Структуры передаются по значению, как обычные переменные, а не по ссылке, как массивы.

Рассмотрим программу, в которой одна функция возвращает структуру, а другая — принимает ее в качестве параметра:

```
#include <stdio.h>  
#include <math.h>  
  
struct circle { int x, y; float dia; char color[10]; };  
struct circle new_circle();  
void cross (struct circle);
```

```

main () {
    struct circle a;

    a = new_circle();
    cross(a);
}

struct circle new_circle() {
    struct circle new;

    printf("Координаты: "); scanf("%d%d", &new.x, &new.y);
    printf("Диаметр: "); scanf("%f", &new.dia);
    printf("Цвет: "); scanf("%s", new.color); //gets(new.color);

    return new;
}

void cross (struct circle c) {
    double hyp;

    hyp = sqrt((double) c.x * c.x + (double) c.y * c.y);
    printf("Расстояние от центра круга до начала координат: %.2lf\n", hyp);
    if (hyp <= c.dia / 2) puts("Круг пересекает начало координат");
    else puts("Круг не содержит точки начала координат");
}

```

Примечание. При компиляции программы в GNU/Linux команда выглядит так: `gcc program.c -lm`. Это связано с использованием библиотеки с математическими функциями.

- Объявляется структура `circle` как глобальный тип данных. Таким образом любая, а не только `main()`, функция может создавать переменные этого типа.
- Функция `new_circle()` возвращает структуру, а функция `cross()` принимает структуру по значению. Следует отметить, что можно создавать функции, которые как принимают (возможно, несколько структур) так и возвращают структуру.
- В функции `new_circle()` создается переменная `new` типа `struct circle`, поля которой заполняются пользователем. Функция возвращает значение переменной `new` в функцию `main()`, где это значение присваивается переменной `a`, которая также принадлежит типу `struct circle`.
- Функция `cross()` определяет, пересекает ли круг начало координат. В ее теле вычисляется расстояние от центра круга до начала координат. Это расстояние является гипотенузой прямоугольного треугольника, длина катетов которого равна значениям `x` и `y`. Далее, если гипотенуза меньше радиуса, то круг пересекает начало координат, т.е. точку $(0, 0)$.
- В функции `main()` при вызове `cross()` данные, содержащиеся в переменной `a`, копируются и присваиваются переменной `c`.

Указатели и структуры

В отличие от массивов, структуры передаются в функции по значению. Это не всегда рационально, т.к. структуры могут быть достаточно большого размера, и копирование таких участков памяти может замедлять работу программы. Поэтому часто структуры в

функцию передают по ссылке, при этом можно использовать как указатель, так и операцию получения адреса.

```
struct book new; // переменная-структура
struct book *pnew; // указатель на структуру
reader(&new); // передаем адрес
pnew = &new;
reader(pnew); // передаем указатель
```

Тогда функция reader() должна иметь примерно такое объявление:

```
void reader (struct book *pb);
```

Возникает вопрос, как при использовании указателей обращаться к элементам структур? Во первых надо получить саму структуру, т.е. если pnew указатель, то сама структура будет *pnew. Далее можно уже обращаться к полям через точку: *pnew.title. Однако это выражение не верно, т.к. приоритет операции "точка" (обращение к полю) выше операции "звездочка" (получить значение по адресу). Таким образом приведенная запись сначала пытается получить значение поля title у указателя pnew, но у pnew нет такого поля. Проблема решается с помощью скобок, которые изменяют порядок выполнения операций: (*pnew).title. В таком случае сначала извлекается значение по адресу pnew, это значение является структурой. Затем происходит обращение к полю структуры.

В языке программирования С записи типа (*pnew).title часто заменяют на такие: pnew->title, что позволяет синтаксис языка. Когда в программе вы видите стрелку (тире и скобка) всегда помните, то, что написано до стрелки, — это указатель на структуру, а не переменная-структура.

Пример кода с использованием указателей:

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

```
struct circle { int x, y; float dia; };
void inversion (struct circle *);

main () {
    struct circle cir, *pc = &cir;

    pc->x = 10; pc->y = 7; pc->dia = 6;
    inversion(pc);
    printf("x = %d, y = %d\n", cir.x, cir.y);
}

void inversion(struct circle *p) {
    p->x = -p->x;
    p->y = -p->y;
}
```

Массивы структур

Обычно создание в программе одной переменной структурного типа не имеет особого смысла. Чаще структурами пользуются, когда необходимо описать множество похожих объектов, имеющих разные значения признаков. Значения каждого объекта следует объединить вместе (в структуру) и тем самым отделить от значений других объектов. Например, описание ряда книг или множества людей. Таким образом мы можем

организовать массив, где каждый элемент представляет собой отдельную структуру, а все элементы принадлежат одному и тому же структурному типу.

Напишем программу для учета персональных компьютеров в организации. Каждая структура будет описывать определенные модели и содержать поле, в котором будет указано количество таких объектов. Поэтому при объявлении структурного типа данных следует описать такие поля как тип компьютера, модель процессора, количество.

Программа будет предоставлять возможность получать информацию о всех моделях и изменять количество компьютеров указанной пользователем модели. В программе будут определены две функции (помимо `main()`): для вывода всей информации и для изменения количества компьютеров.

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

```
#define N 5
```

```
struct computer { char *type; char *proc; int qty; };
void viewer (struct computer *);
void changer (struct computer *);

main () {
    struct computer comps[N]= {
        "Desktop", "earlier than P4", 10,
        "Desktop", "P4", 30 ,
        "Desktop", "Core", 20 ,
        "Desktop", "AMD", 2 ,
        "Notebook", "Core", 1 };

    viewer(comps);
    changer(comps);
    viewer(comps);
}

void viewer (struct computer *comp) {
    int i;

    for (i=0; i < N; i++, comp++)
        printf("%2d % -8s - % -15s: %3d\n", i+1, comp->type, comp->proc, comp->qty);
}

void changer (struct computer *comp) {
    int i, differ;

    printf("Введите номер модели: ");
    scanf("%d", &i); i--;
    printf("На сколько уменьшить или увеличить: ");
    scanf("%d", &differ);
    (comp+i)->qty += differ;
}
```

- Массив структур инициализируется при его объявлении.

- Функции `viewer()` и `changer()` принимают указатели на структуру `computer`.
- В теле `viewer()` указатель инкрементируется в заголовке цикла; таким образом указывая на следующий элемент массива, т.е. на следующую структуру.
- В выражении `(comp+i)->qty` скобки необходимы, т.к оператор `->` имеет более высокий приоритет. Скобки позволяют сначала получить указатель на *i*-ый элемент массива, а потом обратиться к его полю.
- Декрементирование *i* в функции `changer()` связано с тем, что индексация начинается с нуля, а номера элементов массива, которые пользователь видит на экране, с единицы.
- Для того, чтобы уменьшить количество компьютеров, при запросе надо ввести отрицательное число.

Пример результата работы программы:

```

1) Desktop - earlier than P4: 10
2) Desktop - P4 : 30
3) Desktop - Core : 20
4) Desktop - AMD : 2
5) Notebook - Core : 1
Введите номер модели: 5
На сколько уменьшить или увеличить: 3
1) Desktop - earlier than P4: 10
2) Desktop - P4 : 30
3) Desktop - Core : 20
4) Desktop - AMD : 2
5) Notebook - Core : 4

```

Объединения - это объект, позволяющий нескольким переменным различных типов занимать один участок памяти. Объявление объединения похоже на объявление структуры:

```
union union_type {
    int i;
    char ch;
};
```

Как и для структур, можно объявить переменную, поместив ее имя в конце определения или используя отдельный оператор объявления. Для объявления переменной `cvt` объединения `union_type` следует написать:

```
union union_type cvt;
```

В `cvt` как целое число *i*, так и символ *ch* занимают один участок памяти. (Конечно, *i* занимает 2 или 4 байта, а *ch* — только 1.) Рисунок показывает, как *i* и *ch* разделяют один участок памяти (предполагается наличие 16-битных целых). Можно обратиться к данным, сохраненным в `cvt`, как к целому числу, так и к символу.

Рисунок: Использование переменными *i* и *cvt*, (размер переменной целого типа принимается по умолчанию)

Когда объявлено объединение, компилятор автоматически создает переменную достаточного размера для хранения наибольшей переменной, присутствующей в объединении.

Для доступа к членам объединения используется синтаксис, применяемый для доступа к структурам - с помощью операторов «точка» и «стрелка». Чтобы работать с объединением напрямую, надо использовать оператор «точка». Если к переменной объединения обращение происходит с помощью указателя, надо использовать оператор

«стрелка». Например, для присваивания целого числа 10 элементу i объединения cnvt следует написать:

```
cnvt.i = 10;
```

Использование объединений помогает создавать машинно-независимый (переносимый) код. Поскольку компилятор отслеживает настоящие размеры переменных, образующих объединение, уменьшается зависимость от компьютера. Не нужно беспокоиться о размере целых или вещественных чисел, символов или чего-либо еще.

Объединения часто используются при необходимости преобразования типов, поскольку можно обращаться к данным, хранящимся в объединении, совершенно различными способами. Рассмотрим проблему записи целого числа в файл. В то время как можно писать любой тип данных (включая целый) в файл с помощью fwrite(), для данной операции использование fwrite() слишком «жиরно». Используя объединения, можно легко создать функцию, побайтно записывающую двоичное представление целого в файл. Хотя существует несколько способов создания данной функции, имеется один способ выполнения этого с помощью объединения. В данном примере предполагается использование 16-битных целых. Объединение состоит из одного целого и двухбайтного массива символов:

```
union pw {  
    int i;  
    char ch[2];  
};
```

Объединение позволяет осуществить доступ к двум байтам, образующим целое, как к отдельным символам. Теперь можно использовать pw для создания функции write_int(), показанной в следующей программе:

```
#include <stdio.h>  
#include <stdlib.h>  
union pw {  
    int i;  
    char ch[2];  
};  
  
int write_int(int num, FILE *fp);  
  
int main()  
{  
    FILE *fp;  
    fp = fopen("test.tmp", "w+");  
  
    if(fp==NULL) {  
        printf("Cannot open file. \n");  
        exit(1);  
    }  
  
    write_int(1000, fp);  
    fclose(fp);  
    return 0;  
}
```

```
/* вывод целого с помощью объединения */  
  
int write_int (int num, FILE *fp) {  
union pw wrd;  
wrd.i = num;  
putc(wrd.ch[0], fp); /* вывод первой половины */  
return putc(wrd.ch[1], fp); /* вывод второй половины */  
}
```

Хотя `write_int()` вызывается с целым, она использует объединение для записи обеих половинок целого в дисковый файл побайтно.

15. Понятие файла. Файловая система. Операции с файлом.

Информация, представляемая для обработки на компьютере, называется данными. Для хранения на устройствах внешней памяти данные организуют в виде файлов. Файл — именованная область внешней памяти.

Способ организации как служебной, так и пользовательской информации о файлах на носителях называют файловой системой. Конкретная файловая система определяет, в частности, правила именования файлов.

Необходимые для выполнения операций с файлами и носителями программные средства входят в состав операционных систем. Такие программные средства не изменяют и не обращаются к содержимому файлов, а оперируют с ними просто как с целым, непрерывным массивом данных. Таким образом, файловая система обеспечивает выполнение операций для любых программ.

Имя файлу присваивает пользователь, или программа, создающая файл, предлагает имя в автоматическом режиме. По историческим причинам для пользователя имя файла в операционных системах фирмы Майкрософт состоит из двух частей, разделенных точкой: собственно имени и расширения. Тип файла определяется по его расширению, которое задает программа, сохраняющая файл.

С точки зрения прикладных программ, файл представляет собой некоторую последовательность байтов. Используя такой подход, как доступ к файлам, организуется также доступ к некоторым устройствам, которые принимают или возвращают поток байтов. К таким устройствам относятся принтеры, модемы, клавиатура или поток текстового вывода на экран и др.

В некоторых операционных системах предусмотрен такой доступ и к служебной информации самих носителей. Для работы с такими файлами предусмотрены специальные, зарезервированные системой, имена файлов.

Следует помнить, что для ОС линии Microsoft:

между именем и расширением ставится точка, не входящая ни в имя, ни в расширение;

имя файла можно набирать в любом регистре, т.к. для системы все буквы строчные;

символы, не использующиеся в имени файла * = + [] \ ; : , . < > / ?

имена устройств не могут использоваться в качестве имён файлов (prn, lpt, com, con, nul).

Наиболее часто встречающиеся расширения:

EXE, COM — готовая к выполнению программа;
BAT — пакетный командный файл;
SYS — программа-драйвер устройства (системная);
BAK — резервная копия файла;
OBJ — объектный модуль («полуфабрикат» программы);
DAT — файл данных со служебной информацией;
BAS — исходный текст программы на Бейсике;
TXT — текстовый файл;
DOC — документ, созданный в Microsoft Word.

Для удобства хранения и работы файловые структуры организуются с помощью вложенных каталогов (папок).

Каталог — специальный системный файл, в котором хранится служебная информация о файлах.

На каждом носителе может быть множество каталогов. В каждом каталоге может быть зарегистрировано много файлов, но каждый файл регистрируется только в одном каталоге

На каждом логическом томе присутствует один главный, или корневой, каталог. В нем регистрируются файлы и подкаталоги (каталоги 1 уровня). В каталогах 1 уровня регистрируются файлы и каталоги 2 уровня и т. д. Получается древовидная структура каталогов, например:

Каталог, с которым работает пользователь в настоящий момент, называется текущим.

Когда используется файл не из текущего каталога, программе, осуществляющей доступ к файлу, необходимо указать, где именно этот файл находится. Это делается с помощью указания пути к файлу.

Путь к файлу — это последовательность имен каталогов, в операционных системах Windows разделенных символом «\» (в ОС линии UNIX используется символ «//»). Этот путь задает маршрут к тому каталогу, в котором находится нужный файл.

Рассмотрим, например, запись \KLASS10\DOCS\START2\text.doc

Она означает, что файл text.doc находится в подкаталоге START2, который находится в каталоге DOCS, а он в свою очередь находится в каталоге KLASS10 корневого каталога.

Над файлами можно производить следующие основные операции: копирование, перемещение, удаление, переименование и пр.

Каждый файл на диске имеет свой адрес. Чтобы понять принцип доступа к информации, хранящейся в файле, необходимо знать способ записи данных на носители информации.

Перед использованием диск размечается на дорожки и секторы (форматируется). С точки зрения оборудования разметка — это процесс записи на носитель служебной информации, отмечающей конец и начало каждого сектора. Обычный объем сектора —

512 байт. На одной стороне размещается 80 дорожек. Каждая дорожка содержит 18 секторов.

Названия «сектор», «дорожка» введены для дисковых носителей. Во многих современных носителях информации, использующих хранение данных в энергонезависимой памяти, эти понятия поддерживаются реализацией файловых систем для обеспечения общих принципов работы.

В одной из распространенных файловых систем, FAT, предусматривается, что все файлы перечислены в каталогах. Обязателен корневой каталог, размещенный в определенном месте диска. О каждом из перечисленных в каталоге файлов помимо обычных данных известно местоположение (в виде номера) начала файла.

Для того, чтобы определить, какие именно секторы занимает файл, применяется второй обязательный элемент файловой системы — таблица FAT (размещения файлов).

Таблица представляет собой массив ячеек. Размер ячейки фиксирован и отражается в номере файловой системы (12, 16, 32 бита). Каждый файл занимает некоторую последовательность секторов, не обязательно последовательно расположенных. При сохранении файла в ячейку записывается номер следующего сектора в цепочке.

Поскольку на современных дисках секторов существенно больше, чем можно записать номеров в таблице, то секторы объединяют в кластеры. Именно кластерами и распределяется пространство на дисках, в результате эта файловая система неэффективно работает с мелкими файлами.

Сделать эту проблему менее острой позволяет увеличение размера ячейки в FAT. Это позволяет уменьшить размер кластера и увеличить количество адресов (файлов) на диске. В операционных системах, начиная с Windows 98, реализована FAT-32.

Помимо этой файловой системы, существует большое количество других, разработанных для разных операционных систем и решаемых задач.

16. Работа с файлами в языке СИ.

Для программиста открытый файл представляется как последовательность считываемых или записываемых данных. При открытии файла с ним связывается **поток ввода-вывода**. Выводимая информация записывается в поток, вводимая информация считывается из потока.

Когда поток открывается для ввода-вывода, он связывается со стандартной структурой типа FILE, которая определена в **stdio.h**. Структура FILE содержит необходимую информацию о файле.

Открытие файла осуществляется с помощью функции **fopen()**, которая возвращает указатель на структуру типа FILE, который можно использовать для последующих операций с файлом.

FILE *fopen(name, type);

name — имя открываемого файла (включая путь),
type — указатель на строку символов, определяющих способ доступа к файлу:

- **«r»** — открыть файл для чтения (файл должен существовать);
- **«w»** — открыть пустой файл для записи; если файл существует, то его содержимое теряется;
- **«a»** — открыть файл для записи в конец (для добавления); файл создается, если он не существует;

- «**r+**» — открыть файл для чтения и записи (файл должен существовать);
- «**w+**» — открыть пустой файл для чтения и записи; если файл существует, то его содержимое теряется;
- «**a+**» — открыть файл для чтения и дополнения, если файл не существует, то он создаётся. Возвращаемое значение — указатель на открытый поток. Если обнаружена ошибка, то возвращается значение NULL.

Функция `fclose()` закрывает поток или потоки, связанные с открытыми при помощи функции `fopen()` файлами. Закрываемый поток определяется аргументом функции `fclose()`. Возвращаемое значение: значение 0, если поток успешно закрыт; константа EOF, если произошла ошибка.

```
#include <stdio.h>
int main() {
    FILE *fp;
    char name[] = «my.txt»;
    if(fp = fopen(name, «r») != NULL) { // открыть файл удалось?
        ... // требуемые действия над данными
    } else printf(«Не удалось открыть файл»);
    fclose(fp);
    return 0;
}
```

Чтение символа из файла:

```
char fgetc(поток);
```

Аргументом функции является указатель на поток типа FILE. Функция возвращает код считанного символа. Если достигнут конец файла или возникла ошибка, возвращается константа EOF.

Запись символа в файл:

```
fputc(символ, поток);
```

Аргументами функции являются символ и указатель на поток типа FILE. Функция возвращает код считанного символа.

Функции `fscanf()` и `fprintf()` аналогичны функциям `scanf()` и `printf()`, но работают с файлами данных, и имеют первый аргумент — указатель на файл.

```
fscanf(поток, «ФорматВвода», аргументы);
fprintf(поток, «ФорматВывода», аргументы);
```

Функции `fgets()` и `fputs()` предназначены для ввода-вывода строк, они являются аналогами функций `gets()` и `puts()` для работы с файлами.

```
fgets(УказательНаСтроку, КоличествоСимволов, поток);
```

Символы читаются из потока до тех пор, пока не будет прочитан символ новой строки «**\n**», который включается в строку, или пока не наступит конец потока EOF или не будет прочитано максимальное количество символов. Результат помещается в указатель на строку и заканчивается нуль- символом «**\0**». Функция возвращает адрес строки.

```
fputs(УказательНаСтроку, поток);
```

Копирует строку в поток с текущей позиции. Завершающий нуль- символ не копируется.

Пример Ввести число и сохранить его в файле s1.txt. Считать число из файла s1.txt, увеличить его на 3 и сохранить в файле s2.txt.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *S1, *S2;
    int x, y;
    system(«chcp 1251»);
    system(«cls»);
    printf(«Введите число: »);
```

```

scanf(<<%d>>, &x);
S1 = fopen(<<S1.txt>>, <<w>>);
fprintf(S1, <<%d>>, x);
fclose(S1);
S1 = fopen(<<S1.txt>>, <<r>>);
S2 = fopen(<<S2.txt>>, <<w>>);
fscanf(S1, <<%d>>, &y);
y += 3;
fclose(S1);
fprintf(S2, <<%d\n>>, y);
fclose(S2);
return 0;
}

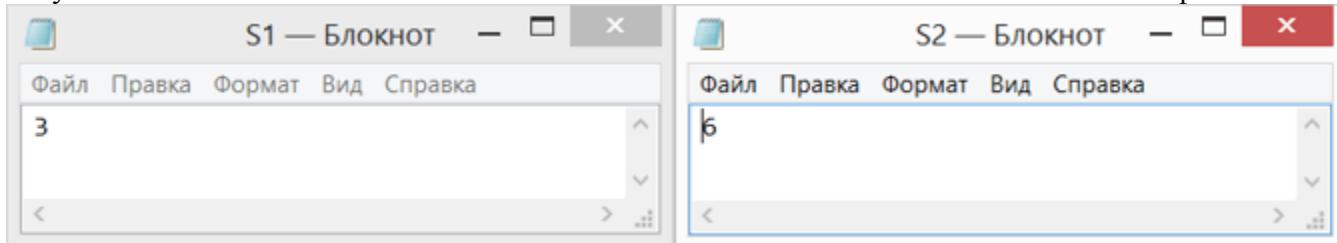
```

Результат

выполнения

2

файла



Другие подходы для работы с файлами

File descriptors. Open, close, read, write

В языке С есть много способов работы с файлами. Помимо структуры FILE можно использовать так называемые дескрипторы файла (file descriptors). Дескриптор файла -- целое неотрицательное число. Оно обозначает номер открытого файла в таблице открытых файлов операционной системы. Использование дескрипторов файла -- более низкий уровень, чем нежели использование структуры FILE. Структура FILE -- сущность языка С и его стандартной библиотеки, тогда как дескриптор файла -- сущность операционной системы. Например, при работе со структурой FILE автоматически создается буфер, и программист работает с более высокоуровневой абстракцией. А при работе с дескрипторами файла программист должен позаботится о буферизации вручную.

Пример работы с дескрипторами файла довольно прост и почти в точности повторяет процесс работы со структурой FILE:

```
int fd = open("...");
```

Сходство работы с дескрипторами файла с работой со структурой FILE заключается в том, что в названии функций отсутствует буква "f". Иногда параметры функций незначительно отличаются.

Структуру FILE полезно использовать при работе с настоящими "файлами" (которые находятся на жестком диске). Использовать дескрипторы файла полезно в случаях работы со специальными "файлами". В этом подходе есть своя специфика работы, но сейчас просто полезно знать, что такой подход существует.

Аналогами stdin, stdout и stderr в дескрипторах файла являются числа 0, 1 и 2 соответственно. Стандарт POSIX.1 обозначил числа 0, 1, 2 символическими константами STDIN_FILENO, STDOUT_FILENO и STDERR_FILENO соответственно.

Memory mapping. Что делает mmap

Следующий способ работы с файлами удобен в тех случаях, когда приходится читать файл нелинейно: надо "ходить" вперед и назад. В предыдущих подходах такие ситуации оказывались неудобными с точки зрения программирования: получился бы громоздкий код.

В языке C был придуман удобный способ работы в таких ситуациях, который называется memory mapping. Соответствующая функция:

```
char *ptr = mmap("...");
```

Работает эта функция примерно так. Мы указываем этой функции файл на диске, и она "отображает" этот файл в такую-то область в памяти. В результате работы функции мы получаем указатель на начало файла. И потом мы можем работать с этим файлом как с обычным указателем на какую-то область памяти: можем "ходить" вперед и назад по этому файлу.

Можно "отобразить" не весь файл целиком, а, например, отдельную часть файла: с 3-его килобайта по 4-ый килобайт.

17. Отладка компьютерных программ. Опции компилятора. Понятие трассировки. Управление точками останова. Команды отладчика.

Отладка — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла ошибка, приходится:

- узнавать текущие значения переменных;
- выяснить, по какому пути выполнялась программа.

Существуют две взаимодополняющие технологии отладки.

- Использование отладчиков — программ, которые включают в себя пользовательский интерфейс для пошагового выполнения программы: оператор за оператором, функция за функцией, с остановками на некоторых строках исходного кода или при достижении определённого условия.
- Вывод текущего состояния программы с помощью расположенных в критических точках программы операторов вывода — на экран, принтер, громкоговоритель или в файл.

GCC - это свободно доступный оптимизирующий компилятор для языков C, C++.

Программа **gcc**, запускаемая из командной строки, представляет собой надстройку над группой компиляторов. В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, **gcc** запускает необходимые препроцессоры, компиляторы, линкеры.

Файлы с расширением **.cc** или **.C** рассматриваются, как файлы на языке C++, файлы с расширением **.c** как программы на языке C, а файлы с расширением **.o** считаются объектными.

Чтобы откомпилировать исходный код C++, находящийся в файле **F.cc**, и создать объектный файл **F.o**, необходимо выполнить команду:

```
gcc -c F.cc
```

Опция **-c** означает «только компиляция».

Чтобы скомпоновать один или несколько объектных файлов, полученных из исходного кода - **F1.o**, **F2.o**, ... - в единый исполняемый файл **F**, необходимо ввести команду:

```
gcc -o F F1.o F2.o
```

Опция **-o** задает имя исполняемого файла.

Можно совместить два этапа обработки - компиляцию и компоновку - в один общий этап с помощью команды:

```
gcc -o F <compile-and-link-options> F1.cc ... -lg++ <other-libraries>
```

<compile-and-link –options> - возможные дополнительные опции компиляции и компоновки. Опция **-lg++** указывает на необходимость подключить стандартную библиотеку языка C++, **<other-libraries>** - возможные дополнительные библиотеки. После компоновки будет создан исполняемый файл F, который можно запустить с помощью команды **./F <arguments>**. Стока **<arguments>** определяет аргументы командной строки Вашей программы.

В процессе компоновки очень часто приходится использовать библиотеки. Библиотекой называют набор объектных файлов, сгруппированных в единый файл и проиндексированных. Когда команда компоновки обнаруживает некоторую библиотеку в списке объектных файлов для компоновки, она проверяет, содержат ли уже скомпонованные объектные файлы вызовы для функций, определенных в одном из файлов библиотек. Если такие функции найдены, соответствующие вызовы связываются с кодом объектного файла из библиотеки. Библиотеки могут быть подключены с помощью опции вида **-lname**. В этом случае в стандартных каталогах, таких как **/lib**, **/usr/lib**, **/usr/local/lib** будет проведен поиск библиотеки в файле с именем **lname.a**. Библиотеки должны быть перечислены после исходных или объектных файлов, содержащих вызовы к соответствующим функциям.

Опции компиляции

Среди множества опций компиляции и компоновки наиболее часто употребляются следующие:

Опция	Назначение
-c	Эта опция означает, что необходима только компиляция. Из исходных файлов программы создаются объектные файлы в виде name.o . Компоновка не производится.
-Dname=value	Определить имя name в компилируемой программе, как значение value . Эффект такой же, как наличие строки #define name value в начале программы. Часть =value может быть опущена, в этом случае значение по умолчанию равно 1.
-o file-name	Использовать file-name в качестве имени для создаваемого файла.
-lname	Использовать при компоновке библиотеку libname.so
-Llib-path -Iinclude-path	Добавить к стандартным каталогам поиска библиотек и заголовочных файлов пути lib-path и include-path соответственно.
-g	Поместить в объектный или исполняемый файл отладочную информацию для отладчика gdb . Опция должна быть указана и для компиляции, и для компоновки. В сочетании -g рекомендуется использовать опцию отключения оптимизации -O0 (см.ниже)
-MM	Вывести зависимости от заголовочных файлов , используемых в Си или C++ программе, в формате, подходящем для утилиты make . Объектные или исполняемые файлы не создаются.
-pg	Поместить в объектный или исполняемый файл инструкции профилирования для генерации информации, используемой утилитой gprof . Опция должна быть указана и для компиляции, и для компоновки. Собранная с опцией -pg программа при запуске генерирует файл статистики. Программа gprof на основе этого файла создает расшифровку, указывающую время, потраченное на выполнение каждой функции.
-Wall	Вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы.
-O1 -O2 -O3	Различные уровни оптимизации.
-O0	Не оптимизировать. Если вы используете многочисленные -O опции с номерами или без номеров уровня, действительной является последняя такая опция.
-I	Используется для добавления ваших собственных каталогов для поиска заголовочных файлов в процессе сборки
-L	Передается компоновщику. Используется для добавления ваших собственных каталогов для поиска библиотек в процессе сборки.
-l	Передается компоновщику. Используется для добавления ваших

собственных библиотек для поиска в процессе сборки.

Трассировка — процесс пошагового выполнения программы. В режиме трассировки программист видит последовательность выполнения команд и значения переменных на данном шаге выполнения программы, что позволяет легче обнаруживать ошибки.

Трассировка может быть начата и окончена в любом месте программы, выполнение программы может останавливаться на каждой команде или на точках останова, трассировка может выполняться с заходом в процедуры и без заходов, а также осуществляться в обратном порядке (шаг назад).

Краткое описание отладчика GDB

Основные возможности отладчика GDB (GNU Debugger):

- символьная отладка программ как в терминах языков высокого уровня (СИ, СИ++, Модула-2), так и языка ассемблера (уровень машинных команд);
- запуск программ под управлением отладчика, анализ посмертного дампа (согласно файла) аварийно завершившихся программ, "подключение" к уже выполняющейся программе;
- пошаговое выполнение программ;
- установка точек останова (возможно, условных) в программе;
- просмотр стеков вложенных вызовов функций (процедур);
- установка точек слежения за переменными программы;
- изменение естественного хода вычислительного процесса в программе;
- изменение исполняемого кода программы;
- работа практически во всех распространенных универсальных ОС и на всех машинных архитектурах
- кроссп-отладка программ.

Отладчик GDB является свободно распространяемым ПО организации [GNU](#). Интерфейс с пользователем организован "через командную строку", но имеется развитая свободно распространяемая графическая оболочка [DDD](#) (Data Display Debugger).

Ниже дается краткое описание наиболее часто используемых команд отладчика GDB.

Компиляция программы

gcc -g -o prog prog.c

prog.c - имя файла с исходным текстом программы;

prog - имя исполняемого файла программы;

-g - опция включения в исполняемый файл информации для символьной отладки.

Запуск отладчика

gdb prog

отладка программы *prog* под управлением отладчика

gdb prog core

анализ посмертного дампа программы *prog*

gdb prog pid

"подключение" отладчика к уже выполняющейся программе *prog* с

идентификатором процесса *pid*

gdb -h

справка о всех возможных опциях отладчика

Команды отладчика

- Команды отладчика вводятся в ответ на приглашение (**gdb**).
- Команда состоит из имени и (необязательно) нескольких аргументов. Разделителем служит один или более пробелов.
- Имя любой команды может быть сокращено справа до уникальной последовательности букв. Имена наиболее часто используемых команд могут быть сокращены до одной начальной буквы без коллизий с именами других команд.
- Ввод пустой строки в качестве команды инициирует повторение последней выполненной команды.
- Клавиши "стрелки" могут быть использованы для просмотра истории ранее выполненных команд и их редактирования.
- Клавиша табуляции TAB может быть использована для дополнения любых слов, составляющих команду. Если после нажатия TAB раздается звуковой сигнал, то это означает, что существует несколько возможностей достроить слово; повторное нажатие TAB выдает список всех возможных вариантов продолжения.
- Подсказку по командам отладчика можно получить командой **help**.
- Команда **quit** завершает работу отладчика.

Выполнение программы

set environment varname value

установить/переустановить переменную среды *varname* в значение *value*

show environment varname

показать значение переменной среды *varname*

unset environment varname

удалить переменную среды *varname*

set args [arg1 ...]

задать аргументы программы

show args

показать аргументы программы

run (или r)

запустить программу на выполнение под управлением отладчика

Доступ к исходному тексту

list (или l) funcname

распечатать порцию исходного текста программы, начиная с функции *funcname*

list (или l) linenum

распечатать порцию исходного текста программы вокруг строки с номером *linenum*

list (или l) -

распечатать следующую порцию исходного текста программы

list (или l) -

распечатать предыдущую порцию исходного текста программы

show listsize

показать размер порции распечатываемого текста

set listsize n

установить размер порции распечатываемого текста к величине *n* строк текста

Доступ к машинному коду

disassemble

распечатать машинный код (используя язык ассемблера) функции, включающей в себя адрес текущей точки останова

disassemble *funcname*

распечатать машинный код функции по имени *funcname*

disassemble *address*

распечатать машинный код функции, покрывающей своим кодом адрес *address*

disassemble *s_addr end_addr*

распечатать машинный код в диапазоне адресов *s_addr - end_addr*

info line *linenum*

выдать начальный и конечный адреса машинного кода строки исходного текста с номером *linenum*

info line **address*

выдать номер строки исходного текста, начальный и конечный адреса ее машинного кода для строки, покрывающей адрес *address*

Точки останова (breakpoints)

В программировании **точка останова** (*breakpoint*) — это преднамеренное прерывание выполнения программы, при котором выполняется вызов отладчика (одновременно с этим программа сама может использовать точки останова для своих нужд). После перехода к отладчику программист может исследовать состояние программы (логи, состояние памяти, регистров процессора, стека и т. п.), с тем чтобы определить, правильно ли ведёт себя программа. После остановки в отладчике программа может быть завершена либо продолжена с того же места, где произошёл останов.

На практике точка останова определяется как одно или несколько условий, при которых происходит прерывание программы. Наиболее часто используется условие останова при переходе управления к указанной инструкции программы (*instruction breakpoint*). Другое условие останова — операция чтения, записи или изменения указанной ячейки или диапазона ячеек памяти (*data breakpoint* или *watchpoint*).

break *funcname*

установить точку останова в начало функции *funcname*

break *linenum*

установить точку останова в строке номер *linenum* исходного файла

break +*offset***break -*offset***

установить точку останова в строке исходного текста, отстоящей от текущей позиции на *offset* строк вперед/назад

break **address*

установить точку останова на машинную инструкцию по адресу *address*

whatch *expr*

установить слежение за изменением выражения *expr*

condition *bn expr*

связать с точкой останова номер *bn* логическое условие *expr* (записанное по правилам языка СИ/СИ++)

commands *bn*

связать с точкой останова номер *bn* список команд отладчика, выполняемых автоматически при каждом останове

info break [bn]

выдать информацию о точке останова номер *bn*

disable [bn]

сделать неактивной точку останова номер *bn*

enable [bn]

активировать точку останова номер *bn*

delete [bn]

удалить точку останова номер *bn*

*Продолжение после останова***continue (или c)**

продолжить выполнение программы с места последней точки останова

step (или s)

выполнить следующую строку исходного текста программы (даже если эта строка внутри вызываемой функции) и остановиться

step (или s) *n*

выполнить следующих *n* строк исходного текста программы (даже если эти строки внутри вызываемой функции) и остановиться

next (или n) [n]

выполнить следующих *n* строк исходного текста программы (строки вызываемых функций в число *n* не включаются и останова в них не происходит) и остановиться

finish

продолжить выполнение программы и остановиться в *вызывающей* функции на следующем операторе после вызова текущей функции

stepi (или si) [n]

выполнить строго *n* машинных инструкций и остановиться

nexti (или ni) [n]

выполнить *n* машинных инструкций и остановиться (однако, если среди этих инструкций есть вызовы функций, то эти функции выполняются без останова и их инструкции в число *n* не включаются)

*Доступ к стеку вызова функций***backtrace (или bt)**

выдать список вложенных вызовов функций прикладной программы (от функции, в которой произошел останов, до функции main) с указанием аргументов и мест вызова

*Доступ к переменным программы***print (или p) *expr***

выдать значение выражения *expr*, сконструированного по правилам языка СИ/СИ++ с использованием переменных программы; значение выводится в формате, соответствующем типу данных всего выражения

print/f *expr*

выдать значение выражения *expr* в формате *f*

Допустимыми спецификациями формата являются:

x - шестнадцатиричное целое;

d - десятичное целое со знаком;

u - десятичное целое без знака;

o - восьмеричное число;
t - двоичное число (от **two**);
a - адрес;
c - однобайтовый символ;
f - число с плавающей точкой.

print/f

заново выдать последнее значение в формате *f*

Выражение *expr* помимо конструкций языка СИ/СИ++ может включать в себя:

{*type*}*addr*

объект типа *type*, располагающийся в памяти по адресу *addr*, где *addr* может быть любым выражением, имеющим результат в виде целого или указателя

funcname::*static-var*

статическая переменная *static-var* из функции по имени *funcname*

Доступ к памяти

x {[*n*][*f*][*u*] } *addr*

выдать содержимое памяти программы по адресу *addr*, используя следующие спецификации вывода:

f - формат вывода (один символ), используемый в качестве формата вывода команды print ("умолчанием" здесь является **x** - шестнадцатиричное целое длиной в 4 байта);

u - размер выводимого объекта:

b - байт

h - полуслово (2 байта)

w - слово (4 байта)

g - "гигантское" слово (8байт);

n - счетчик количества выводимых объектов (1 - значение "по умолчанию")

Доступ к регистрам

info registers

выводит имена и содержимое (в шестнадцатиричной и десятичной формах) всех регистров *общего назначения*

info all-registers

выводит имена и содержимое всех регистров программы

info registers *regname1* [*regname2* ...]

выводит содержимое регистров с именами *regname1*, *regname2* ...

Примечание. Имена регистров определяются используемым ассемблером эксплуатируемой машины. В GDB принято имена регистров предварять символом "\$" (например, в GDB \$esp означает регистр указателя стека %esp ассемблера фирмы Sun Microsystems для микропроцессора серии X86* фирмы Intel).

Примечание. Конструкция \$*regname* в командах GDB (кроме команды set) трактуется как "содержимое регистра *regname*", например:

x/xw \$ebp+8

выдать в виде шестнадцатиричного целого числа содержимое одного слова (4 байта) памяти, адрес которого определяется как сумма содержимого регистра ebp и числа 8;

x/s *(char *)(\$esp+4)

выдать в символьном виде содержимое последовательности байтов памяти, начальный адрес которых содержится в слове памяти (4 байта), адрес которого вычисляется как сумма содержимого регистра esp и числа 4.

Изменения в программе

set var *varname* = *expr*

присвоить переменной *varname* программы значение выражения *expr*

set \$*regname* = *expr*

присвоить регистру *regname* значение выражения *expr*

set {*type*}*addr* = *expr*

присвоить значение выражения *expr* переменной типа *type* (допустимы типы данных языка СИ), расположенной в памяти по адресу *addr*. Таким способом можно модифицировать исполняемый машинный код программы, если он доступен для записи, и если в начале сеанса отладки была выдана команда set write on.

Прочие полезные команды

file *prog*

exec-file *prog*

начать сеанс отладки программы *prog*

source *filename*

выполнить последовательность команд отладчика из файла с именем *filename*

info variables

выдать информацию о всех переменных программы

ОПЕРАЦИОННЫЕ СИСТЕМЫ

1. Основные функции ОС. Примеры исполнения этих функций на основе современных ОС.

Операционная система (ОС) – комплекс взаимосвязанных программ, выполняющий

следующие функции:

* предоставление пользователям интерфейса к аппаратным средствам;

* рациональное управление ресурсами вычислительной системы.

Основные функции операционной системы:

1. Обмен данными между компьютером и различными периферийными устройствами (терминалами, принтерами, гибкими дисками, жесткими дисками и т.д.). Такой обмен данными называется "ввод/вывод данных".

2. Обеспечение системы организации и хранения файлов.

3. Загрузка программ в память и обеспечение их выполнения.

4. Организация диалога с пользователем.

Первая задача ОС – организация связи, общения пользователя с компьютером в целом и его отдельными устройствами. Такое общение осуществляется с помощью команд, которые в том или ином виде человек сообщает операционной системе. В ранних вариантах операционных систем такие команды просто вводились с клавиатуры в специальную строку. В последующем были созданы программы – оболочки ОС, которые позволяют общаться не только с ОС не только текстовым языком команд, а с помощью меню (в том числе пиктографического) или манипуляций с графическими объектами.

Вторая задача ОС – организация взаимодействия всех блоков компьютера в процессе выполнения программы, которую назначил пользователь для решения задачи. В частности, ОС организует и следит за размещением в оперативной памяти и на диске нужных для работы программы данных, обеспечивает своевременное подключение устройств компьютера по требованию программы и т.п.

Третья задача ОС – обеспечение так называемых системных работ, которые бывает необходимо выполнить для пользователя. Сюда относится проверка, “лечение” и форматирование диска, удаление и восстановление файлов, организация файловой системы и т.п. Обычно такие работы осуществляются с помощью специальных программ, входящих в ОС и называемых утилитами.

Операционная система выполняет роль связующего звена между аппаратурой компьютера, с одной стороны, и выполняемыми программами, а также пользователем, с другой стороны.

ОС обычно хранится во внешней памяти компьютера – на диске. При включении компьютера она считывается с дисковой памяти и размещается в ОЗУ.

Этот процесс называют загрузкой ОС.

В функции ОС входит:

- осуществление диалога с пользователем;
- ввод-вывод и управление данными;
- планирование и организация процесса обработки программ;
- распределение ресурсов (оперативной памяти, процессора, внешних устройств);
- запуск программ на выполнение;
- всевозможные вспомогательные операции обслуживания;
- передача информации между различными внутренними устройствами;
- программная поддержка работы периферийных устройств (дисплея, клавиатуры, принтера и др.).

2. Виды архитектур ядра ОС. Монолитные и микроядерные архитектуры.

Все операции, связанные с процессами, выполняются под управлением той части операционной системы, которая называется ядром. Ядро представляет собой лишь небольшую часть кода операционной системы в целом, однако оно относится к числу наиболее интенсивно используемых компонент системы. По этой причине ядро обычно резидентно размещается в основной памяти, в то время как другие части операционной системы перемещаются во внешнюю память и обратно по мере необходимости.

Монолитное ядро

Монолитное ядро предоставляет богатый набор абстракций оборудования. Все части монолитного ядра работают в одном адресном пространстве. Это такая схема операционной системы, при которой все компоненты её ядра являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путём непосредственного вызова процедур. Монолитное ядро — старейший способ организации операционных систем. Примером систем с монолитным ядром является большинство UNIX-систем.

Достоинства: Скорость работы, упрощённая разработка модулей. Недостатки: Поскольку всё ядро работает в одном адресном пространстве, сбой в одном из компонентов может нарушить работоспособность всей системы. Примеры: Традиционные ядра UNIX (такие как BSD), Linux; ядро MS-DOS, ядро KolibriOS.

Монолитные ядра имеют долгую историю развития и усовершенствования и, на данный момент, являются наиболее архитектурно зрелыми и пригодными к эксплуатации. Вместе с тем, монолитность ядер усложняет их отладку, понимание кода ядра, добавление новых функций и возможностей, удаление «мёртвого», ненужного, унаследованного от предыдущих версий кода. «Разбухание» кода монолитных ядер также повышает требования к объёму оперативной памяти, требуемому для функционирования ядра ОС. Это делает монолитные ядерные архитектуры малопригодными к эксплуатации в системах, сильно ограниченных по объёму ОЗУ, например, встраиваемых системах, производственных микроконтроллерах и т. д.

Микроядро

Микроядро – предоставляет только элементарные функции управления процессами и минимальный набор абстракций для работы с оборудованием. Большая часть работы осуществляется с помощью специальных пользовательских процессов, называемых сервисами. Решающим критерием «микро-ядерности» является размещение всех или почти всех драйверов и модулей в сервисных процессах, иногда с явной невозможностью загрузки любых модулей расширения в собственно микроядро, а также разработки таких расширений.

Достоинства: Устойчивость к сбоям оборудования, ошибкам в компонентах системы. Основное достоинство микроядерной архитектуры — высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядерной операционной системе можно, не прерывая её работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная

архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

Недостатки: Передача данных между процессами требует накладных расходов. Классические микроядра предоставляют лишь очень небольшой набор низкоуровневых примитивов, или системных вызовов, реализующих базовые сервисы операционной системы.

Сервисные процессы (в принятой в семействе UNIX терминологии — «демоны») активно используются в самых различных ОС для задач типа запуска программ по расписанию (UNIX и Windows NT), ведения журналов событий (UNIX и Windows NT), централизованной проверки паролей и хранения пароля текущего интерактивного пользователя в специально ограниченной области памяти (Windows NT). Тем не менее, не следует считать ОС микроядерными только из-за использований такой архитектуры. Примеры: Symbian OS; Windows CE; OpenVMS; Mach, используемый в GNU/Hurd и Mac OS X; QNX; AIX; Minix; ChorusOS; AmigaOS; MorphOS.

3. Порядок загрузки ПК архитектуры x86. Назначение и основные действия выполняемые BIOS, загрузчиком и ядром операционной системы во время загрузки ПК.

Процессы после включения питания

И так начнем: последовательность загрузки компьютера, первое устройство, которое запускается после нажатия кнопки включения компьютера, — блок питания. Если все питающие напряжения окажутся в норме, на системную плату будет подан специальный сигнал Power Good, свидетельствующий об успешном тестировании блока питания и разрешающий запуск компонентов системной платы. После этого чипсет формирует сигнал сброса центрального процессора, по которому очищаются регистры процессора, и он запускается. Упрощенно процессор работает следующим образом: 1. считывает из системной памяти команду, которая записана в ячейке памяти по первоначальному адресу;

2. выполняет эту команду, после чего читает и выполняет следующую команду и т. д.

Таким образом, его работа — последовательно читать и выполнять команды из памяти. Системная память сконфигурирована так, что первая команда, которую считает процессор после сброса, будет находиться в микросхеме BIOS. Последовательно выбирая команды из BIOS, процессор начнет выполнять процедуру самотестирования, или POST.

Процедура POST

Процедура самотестирования POST состоит из нескольких этапов.

1. Первоначальная инициализация основных системных компонентов.
2. Детектирование оперативной памяти, копирование кода BIOS в оперативную память и проверка контрольных сумм BIOS.
3. Первоначальная настройка чипсета.

4. Поиск и инициализация видеоадаптера. Современные видеоадаптеры имеют собственную BIOS, которую системная BIOS пытается обнаружить в специально отведенном сегменте адресов. В ходе инициализации видеоадаптера на экране появляется первое изображение, сформированное с помощью BIOS видеоадаптера.
5. Проверка контрольной суммы CMOS и состояния батарейки. Если контрольная сумма CMOS ошибочна, будут загружены значения по умолчанию.
6. Тестирование процессора и оперативной памяти. Результаты обычно выводятся на экран.
7. Подключение клавиатуры, тестирование портов ввода/вывода и других устройств.
8. Инициализация дисковых накопителей. Сведения об обнаруженных устройствах обычно выводятся на экран.
9. Распределение ресурсов между устройствами и вывод таблицы с обнаруженными устройствами и назначенными для них ресурсами.
10. Поиск и инициализация устройств, имеющих собственную BIOS.
11. Вызов программного прерывания BIOS INT 19h, который ищет загрузочный сектор на устройствах, указанных в списке загрузки.

В зависимости от конкретной версии BIOS порядок процедуры POST может немного раз отличаться, но приведенные выше основные этапы выполняются при загрузке любого компьютера.

Загрузка операционной системы

После того как успешно завершилась процедура POST, вызывается программное прерывание BIOS INT 19h и запускается процедура поиска загрузочного сектора, который может находиться на жестком диске или сменном носителе. Порядок опроса устройств устанавливается с помощью параметров BIOS First Boot Device, Second Boot Device и Third Boot Device.

Рассмотрим пример, когда в системе установлен следующий порядок загрузки: первое загрузочное устройство — дискета (Floppy), второе — CD/DVD и третье — жесткий диск (HDD). 1. Программа BIOS сначала обратится к дисководу и, обнаружив дискету, считает ее первый сектор, проверит, есть ли в нем загрузчик операционной системы, и передаст управление ему. Если дискета была отформатирована как несистемная, то загрузка остановится с вы-водом соответствующего сообщения, например: Non-System disk or disk error. Replace and press any key when ready (Для продолжения загрузки нужно извлечь дискету из дисковода и нажать любую клавишу). 2. Если дискеты в дисководе нет, система обратится к приводу для CD. Обнаружив загрузочный компакт-диск, система будет загружаться с него. Причем загрузочные CD могут выводить в ходе загрузки различные сообщения. Например, диск с дистрибутивом Windows XP выводит Press any key to boot from CD (Нажмите любую клавишу для загрузки с CD), и если не нажимать никаких клавиш, то через несколько секунд система начнет загружаться с устройства, указанного следующим в списке загрузки.

3. Загрузка с HDD начнется, если нет дискеты в дисководе и загрузочного компакт-диска в приводе для CD. В этом случае BIOS обращается к первому физическому сектору диска, откуда считывает таблицу разделов жесткого диска и код главной загрузочной записи (MBR). После этого BIOS заканчивает свою работу и передает управление коду MBR. Жесткий диск может

состоять из одного или нескольких разделов, и один из них должен быть помечен как активный. Программа, содержащаяся в MBR, считывает загрузчик операционной системы, который находится в первом секторе активного раздела, и запускает его. После этого начинают загружаться основные системные файлы.

В таком порядке можно загружаться с любого устройства, не изменяя параметры BIOS. Однако для обычного использования компьютера желательно установить в BIOS Setup первоочередную загрузку с жесткого диска, поскольку в этом случае процесс пойдет быстрее и не нужно будет постоянно проверять, есть ли диски в дисководах.

Если операционная система не загружается, это может быть связано с неправильным порядком загрузки, а также с повреждением системных файлов или загрузочных областей диска. Например, для успешного начала загрузки Windows 2000/ XP/2003 с жесткого диска условия будут такими.

Последовательность загрузки компьютера

1. В первичные устройства для загрузки обязательно должен быть жесткий диск. Если он не указан первым, то нужно извлечь носители из всех дисководов. Если же в системе присутствует несколько жестких дисков, необходимо проверить, чтобы в списке загрузки значился именно тот накопитель, на котором установлена операционная система.

2. В первом секторе жесткого диска должны быть правильная главная загрузочная запись и таблица разделов. Один из разделов должен быть помечен как активный, а в его первом секторе необходимо наличие загрузчика операционной системы.

Нужные данные записываются в загрузочные области диска во время инсталляции операционной системы, а если загрузочные области повреждены, их можно восстановить, загрузившись с помощью консоли восстановления.

3. В корневой папке загрузочного раздела должны находиться следующие файлы:

- * ntldr — загрузчик операционной системы;
- * boot.ini — текстовый файл со сведениями об установленных операционных системах;
- * ntdetect.com — модуль для сбора информации об имеющемся оборудовании;
- * bootsect.dos — файл с копией загрузочного сектора для загрузки Windows 9x, если такая возможность предусмотрена конфигурацией системы;
- * ntbootdd.sys — необязательный файл, но он необходим для использования жестких дисков SCSI, не поддерживаемых BIOS;
- * bootfont — bin — файл шрифта для меню загрузки; без него сообщения программы загрузки нельзя читать на русском языке.

4. По пути, указанному в файле boot.ini, должна находиться папка с установленной копией Windows и со всеми необходимыми системными файлами.

4. Порядок загрузки ПК архитектуры x86. Трансляция адреса из логического в физиче-ский в реальном режиме x86 архитектуры.

Реальный режим служит для обеспечения преемственности предыдущих процессоров и программ, а также для некоторых применений.

Защищенный режим служит для того, чтобы дать каждой программе несколько независимых защищенных адресных пространств. Страницный режим используется прежде всего для поддержки виртуальной памяти, т.е. среды, в которой большие адресные пространства моделируются на базе небольшой области оперативной памяти и некоторой дисковой памяти. Разработчики систем могут использовать как оба этих механизма (2 и 3), так и любой из них. При одновременном выполнении нескольких программ для защиты одних программ от влияния на них других программ можно использовать любой механизм.

Режим системного управления прежде всего предназначен для обеспечения перехода процессора в состояние пониженного энергопотребления (будет рассмотрен позднее). Трансляция адреса в реальном режиме

Родительская категория: Память. Верхний уровень Категория: Трансляция адреса в защищенном режиме в проц-х x86

В реальном режиме сегментный регистр содержит непосредственно компоненту адреса (см. рисунок ниже).

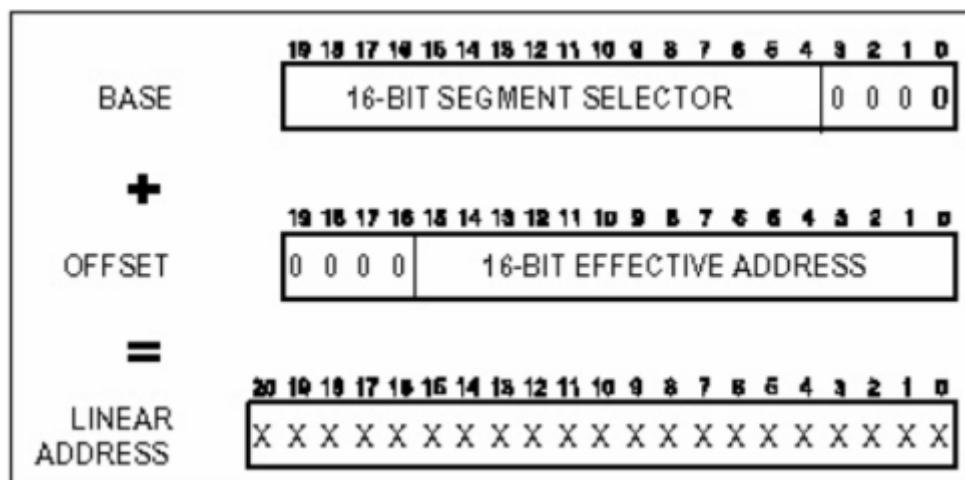


Схема трансляции адреса в реальном режиме

Для получения линейного адреса в реальном режиме содержимое сегментного регистра сдвигается на четыре позиции влево (т.е. умножается на 16) и полученный 20-битовый результат складывается с 16-битовым offset (внутрисегментным смещением), вычисленным в соответствии с используемым способом адресации). Эти действия выполняются в процессоре аппаратно блоком формирования адреса и не заметны для программиста. В каждый данный момент времени активны/ доступны программе (для 386+) шесть сегментов (по числу имеющихся в процессоре сегментных регистров).

В реальном режиме сформированный линейный адрес подается непосредственно на физическую адресную шину, т.е. в реальном режиме понятия линейного и физического адресов совпадают. Таким образом, в реальном режиме размер сегмента фиксирован. Максимальный адрес, который можно сформировать по приведенной схеме, равен $0xFFFF0 + 0xFFFF = 0x10FFEF$, что дает общее количество адресов $0x10FFF0 = 0x10000 + 0x1000 - 0x10 = 220+216-24$.

В реальном режиме содержимое сегментных регистров доступно прикладному программисту и может быть им изменено. Задавая содержимое сегментного регистра, программист может расположить сегмент в физическом адресном пространстве с шагом в 16 байтов.

5. Порядок загрузки ПК архитектуры x86. Трансляция адреса из логического в физический в защищенном режиме x86 архитектуры.

Наиболее полно возможности микропроцессора по адресации памяти реализуются при работе в защищенном режиме. Объем адресуемой памяти увеличивается до 4 Гбайт, появляется возможность страничного режима адресации. Сегменты могут иметь переменную длину от 1 байта до 4 Гбайт.

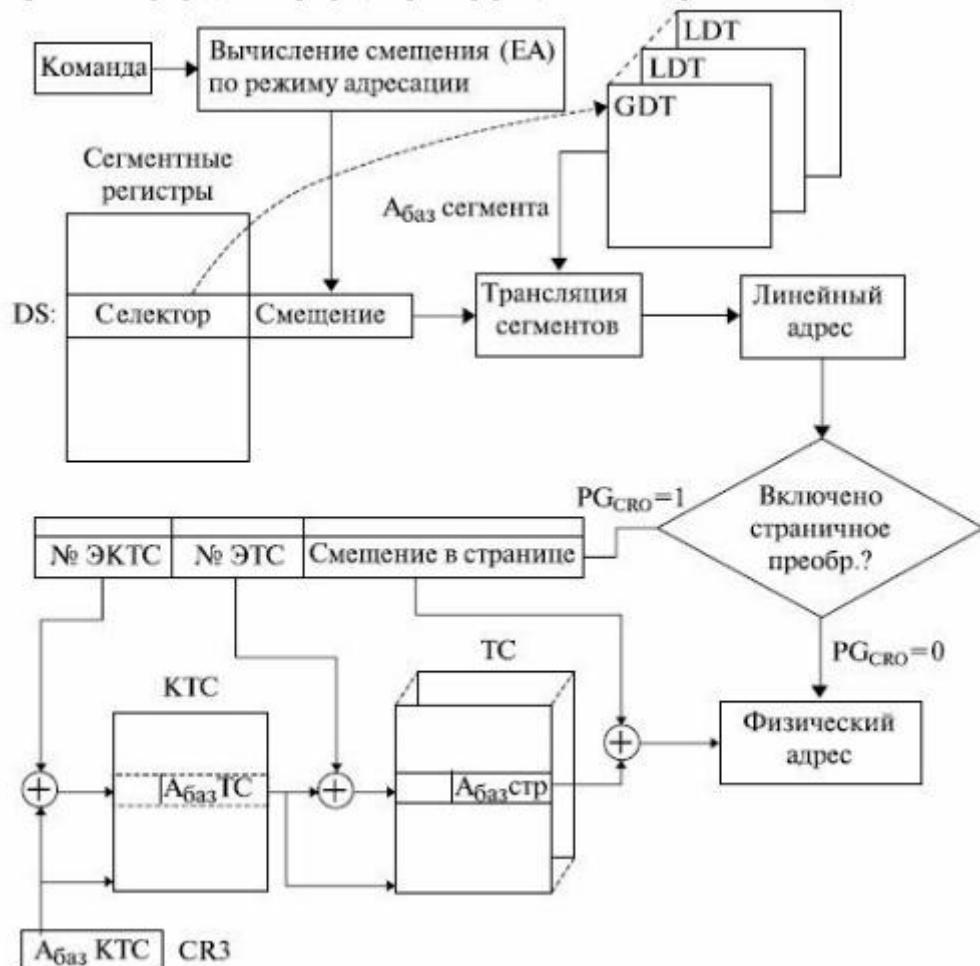


Рисунок: Общая схема формирования физического адреса микропроцессором, работающим в защищенном режиме

Основой формирования физического адреса служит логический адрес. Он состоит из двух частей: селектора и смещения в сегменте.

Селектор содержитится в сегментном регистре микропроцессора и позволяет найти описание сегмента (дескриптор) в специальной таблице дескрипторов. Дескрипторы сегментов хранятся в специальных системных объектах глобальной (GDT) и локальных (LDT) таблицах дескрипторов. Дескриптор играет очень важную роль в функционировании микропроцессора, от формирования физического адреса при различной организации адресного пространства и до организации мультипрограммного режима работы. Поэтому рассмотрим его структуру более подробно.

Сегменты микропроцессора, работающего в защищенном режиме, характеризуются большим количеством параметров. Поэтому в универсальных 32-разрядных микропроцессорах информация о сегменте хранится в специальной 8-байтной структуре данных, называемой дескриптором, а за сегментными регистрами закреплена основная функция - определение местоположения дескриптора.



32-разрядное поле базового адреса позволяет определить начальный адрес сегмента в любой точке адресного пространства в 2^{32} байт (4Гбайт).

Сумма извлеченного из дескриптора начального адреса сегмента и сформированного смещения в сегменте дает линейный адрес (ЛА).

Если в микропроцессоре используется только сегментное представление адресного пространства, то полученный линейный адрес является также и физическим.

Если помимо сегментного используется и страничный механизм организации памяти, то линейный адрес представляется в виде двух полей: старшие разряды содержат номер виртуальной страницы, а младшие смещение в странице. Преобразование номера виртуальной страницы в номер физической проводится с помощью специальных системных таблиц: каталога таблиц страниц (КТС) и таблиц страниц (ТС). Положение каталога таблиц страниц в памяти определяется системным регистром CR3. Физический адрес вычисляется как сумма полученного из таблицы страниц адреса физической страницы и смещения в странице, полученного из линейного адреса.

БОЛЕЕ ПОДРОБНО СТРАНИЦЫ 15-18(26) УЧЕБНИКА 11. Гуров В.В. Архитектура микропроцессоров [Электронный ресурс]/ Гуров В.В.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 115 с.— Режим доступа: <http://www.iprbookshop.ru/56313>.— ЭБС «IPRbooks» **ТАКЖЕ ТАМ ВЫ**

СКОРЕЕ ВСЕГО НАЙДЕТЕ ОТВЕТЫ НА ВОПРОСЫ С 4 ПО +∞ (НА ДРУГИХ СТРАНИЦАХ)

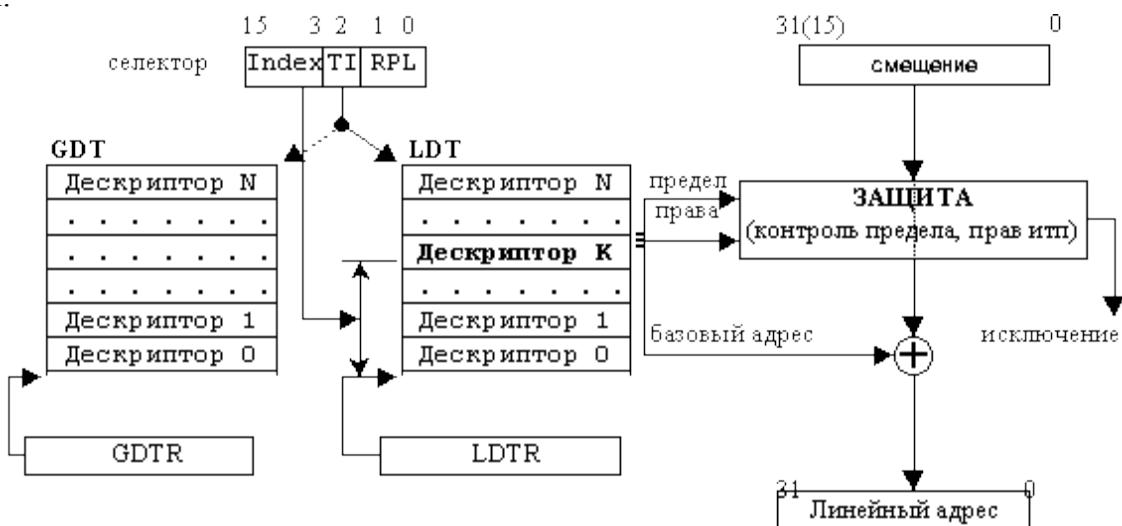
6. Сегментная организация памяти архитектуры x86. Трансляция адреса x86 архитектуры из логического в линейный (виртуальный) адрес.

Логический адрес состоит из двух элементов: селектор сегмента и относительный адрес (смещение). Селектор сегмента может либо находиться непосредственно в коде команды, либо в одном из сегментных регистров. Смещение также может либо непосредственно находиться в коде команды, либо вычисляться на основе значений регистров общего назначения (см. [режимы адресации](#)).

Для вычисления линейного адреса процессор выполняет следующие действия:

- использует селектор сегмента для нахождения дескриптора сегмента;
- анализирует дескриптор сегмента, [контролируя права доступа](#) (сегмент доступен с текущего уровня привилегий) и [предел сегмента](#) (смещение не превышает предел);
- добавляет смещение к базовому адресу сегмента и получает линейный адрес.

Если [страничная трансляция](#) отключена, то сформированный линейный адрес считается физическим и выставляется на шину процессора для выполнения цикла чтения или записи памяти.



Селектор - это 16-битный идентификатор сегмента. Он содержит индекс дескриптора в дескрипторной таблице, бит определяющий, к какой [дескрипторной таблице](#) производится обращение (LDT или GDT), а также запрашиваемые права доступа к сегменту. Если селектор хранится в сегментном регистре, то обращение к дескрипторным таблицам происходит только при загрузке селектора в сегментный регистр, т.к. каждый сегментный регистр хранит соответствующий дескриптор в программно-недоступном ("теневом") регистре-кэше.

Формат селектора:



- Индекс выбирает один из 8192 дескрипторов в таблице дескрипторов. Процессор умножает значение этого индекса на восемь (длину дескриптора) и добавляет

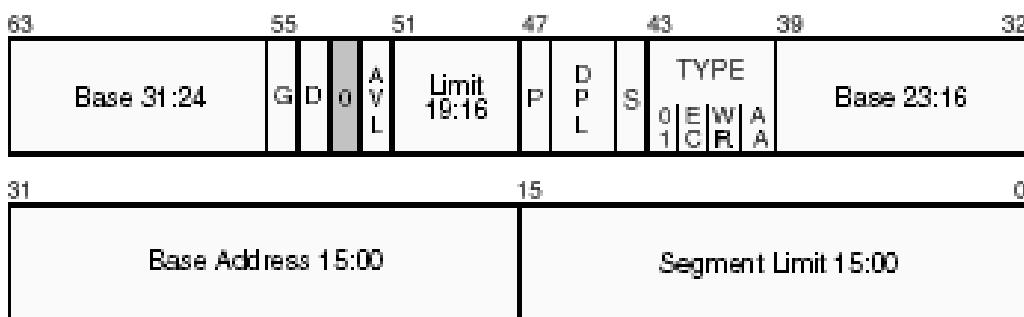
результат к базовому адресу таблицы дескрипторов. Таким образом получается линейный адрес требуемого дескриптора.

- *TI - индикатор таблицы* определяет таблицу дескрипторов, на которую ссылается селектор: TI=0 означает глобальную дескрипторную таблицу (GDT), а TI=1 - используемую в настоящий момент локальную дескрипторную таблицу (LDT).
- *RPL - запрашиваемый уровень привилегий (Requested Privilege Level)*. Используется механизмом защиты.

Первый элемент GDT процессором не используется. Селектор, имеющий нулевые индекс и индикатор таблицы, т.е. селектор, указывающий на первый элемент GDT, называется *нуль-селектором*. Загрузка нуль-селектора в любой регистр сегмента, отличный от регистра CS или SS, не вызывает исключение. Но использование этого регистра сегмента для доступа к памяти вызывает нарушение общей защиты. Это удобно для выявления случайных ссылок.

Для вычисления линейного адреса используются специальные структуры - дескрипторы. *Дескриптор* - это 8-байтная единица описательной информации, распознаваемая устройством управления памятью в защищенном режиме, хранящаяся в дескрипторной таблице. Дескриптор сегмента содержит базовый адрес описываемого сегмента, предел сегмента и права доступа к сегменту. В защищенном режиме сегменты могут начинаться с любого линейного адреса (который называется базовым адресом сегмента) и иметь любой предел вплоть до 4Гбайт.

Формат дескриптора:



- *Базовый адрес сегмента (Base Address)* определяет место сегмента внутри линейного 4Гбайтного адресного пространства. Процессор объединяет три фрагмента базового адреса для формирования одного 32-разрядного значения.
- *Предел сегмента (Segment Limit)* определяет размер сегмента. Задает максимальное (для сегментов стека - минимальное) смещение в сегменте, обращение по которому не вызывает нарушения общей защиты. Процессор связывает две части поля границы для формирования 20-разрядного результата. Затем он интерпретирует поле границы одним из двух способов в зависимости от состояния бита гранулярности: в единицах байт для определения границы до 1Мбайт (G=0); в единицах страниц по 4Кбайт для определения границы до 4Гбайт (G=1), при загрузке поле границы сдвигается влево на 12 бит и младшие биты выставляются в 1.
- *G - бит гранулярности (Granularity)* определяет размер единиц, в которых интерпретируется поле предела. Если G=0, то граница интерпретируется в байтах, иначе в единицах по 4Кбайт.
- *D - размер по умолчанию (Default size)*. Задает разрядность данных по умолчанию для дескрипторов сегментов данных или разрядность команд по умолчанию для дескрипторов сегментов кода: D=0 - 16 бит, D=1 - 32 бит.
- *AVL (available)* - может использоваться по усмотрению системного программиста.

- *P - бит присутствия (Present)*. Если этот бит очищен, то данный дескриптор не может быть использован при трансляции адресов. Когда селектор такого дескриптора загружается в регистр сегмента, процессор переходит к обработке нарушения неприсутствия сегмента.
- *DPL - уровень привилегий дескриптора (Descriptor Privilege Level)* используется механизмом защиты.
- *S - бит системного дескриптора (System)*: определяет, является ли данный сегмент системным ($S=0$) или же сегментом кода/данных ($S=1$).
- *Тип дескриптора (Type)*. Интерпретация этого поля зависит от вида дескриптора. В дескрипторах сегментов кода или данных это поле содержит 3-разрядный тип и один бит флага обращения. Дескрипторы системных сегментов используют все 4 бита.

Типы дескрипторов сегментов:

0 E W A	<u>Сегменты данных</u> E (expand-down) - расширяется вниз (для стека) (writable) - разрешена запись (accessed) - бит обращения
0 0 0 x	сегмент данных, только чтение
0 0 1 x	сегмент данных, чтение-запись
0 1 0 x	сегмент стека, только чтение
0 1 1 x	сегмент стека, чтение-запись
1 C R A	<u>Сегменты кода</u> C (conforming) - подчиняемый (readable) - разрешено чтение (accessed) - бит обращения
1 0 0 x	сегмент кода, только исполнение
1 0 1 x	сегмент кода, исполнение-чтение
1 1 0 x	подчиняемый код, только исполнение
1 1 1 x	подчиняемый код, исполнение-чтение

Флаг обращения (A) устанавливается в дескрипторах сегментов кода и данных при обращении к сегменту. Операционные системы, реализующие виртуальную память на уровне сегментов, могут следить за частотой использования сегмента путем периодической проверки и очистки этого бита.

В качестве сегментов стека используют сегменты данных, доступные для чтения и записи. Попытка загрузить в SS селектор незаписываемого сегмента приведет к нарушению общей защиты. Для динамического изменения размера стека удобно использовать сегменты данных с расширением вниз (бит E=1). Для таких сегментов предел задает минимальный адрес, вызывающий нарушение предела сегмента, поэтому уменьшение предела в таком дескрипторе приведет к добавлению места у вершины стека.

Сегменты кода могут быть либо обычными, либо подчиняемыми (бит C=1). Передача управления между обычными сегментами возможна только на одном уровне привилегий. Для подчиняемых сегментов это правило менее строгое: задача может передать управление на более привилегированный подчиняемый сегмент кода, но при этом он будет выполняться на том же уровне привилегий.

БОЛЕЕ ПОДРОБНО СТРАНИЦЫ 19-21 УЧЕБНИКА 11. Гуров В.В. Архитектура микропроцессоров [Электронный ресурс]/ Гуров В.В.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 115 с.— Режим доступа: <http://www.iprbookshop.ru/56313>.— ЭБС «IPRbooks» **ТАКЖЕ ТАМ ВЫ СКОРЕЕ ВСЕГО НАЙДЕТЕ ОТВЕТЫ НА ВОПРОСЫ С 4 ПО +∞ (НА ДРУГИХ СТРАНИЦАХ)**

7. Страницчная организация памяти архитектуры x86. Трансляция адреса x86 архитектуры из линейного (виртуального) в физический адрес.

Страницчная организация памяти применяется только в защищенном режиме, если в регистре управления CR0 бит PG = 1.

Основное применение страницочного преобразования адреса связано с реализацией виртуальной памяти, которая позволяет программисту использовать большее пространство памяти, чем физическая основная память.

Принцип виртуальной памяти предполагает, что пользователь при подготовке своей программы имеет дело не с физической ОП, действительно работающей в составе компьютера и имеющей некоторую фиксированную емкость, а с виртуальной (кажущейся) одноуровневой памятью, емкость которой равна всему адресному пространству, определяемому размером адресной шины ($L_{шн}$) компьютера:

$$\begin{aligned}V_{\text{вирт}} &>> V_{\text{физ}} \\V_{\text{вирт}} &= 2^{L_{шн}}\end{aligned}$$

Для 32-разрядного микропроцессора:

$$V_{\text{вирт}} = 2^{32} = 4\text{ Гбайт}$$

Программист имеет в своем распоряжении адресное пространство, ограниченное лишь разрядностью адресной шины, независимо от реальной емкости оперативной памяти компьютера и объемов памяти, которые используются другими программами, параллельно обрабатываемыми в мультипрограммной ЭВМ.

На всех этапах подготовки программ, включая загрузку в память, программа представляется в виртуальных адресах, и лишь привыполнении машинной команды виртуальные адреса преобразуются в физические. Для каждой программы, выполняемой вмультпрограммном режиме, создается своя виртуальная память. Каждая программа использует одни и те же виртуальные адреса от нулевого до максимально большого в данной архитектуре.

Для преобразования виртуальных адресов в физические физическая и виртуальная память разбиваются на блоки фиксированной длины, называемые страницами. Объемы виртуальной и физической страниц совпадают. Страницы виртуальной и физической памяти нумеруются. Отсутствующие в физической памяти страницы обычно хранятся во внешней памяти. Фиксированный размер всех страниц позволяет загрузить любую нужную виртуальную страницу в любую физическую.

При страничном представлении памяти виртуальный (логический) адрес представляет собой номер виртуальной страницы и смещение внутри этой страницы. В свою очередь, физический адрес - это номер физической страницы и смещение в ней.

Правила перевода номеров виртуальных страниц в номера физических страниц обычно задаются в виде таблицы страничного преобразования. Такие таблицы формируются системой управления памятью и модифицируются каждый раз при перераспределении памяти. Операционная система постоянно отслеживает состояние виртуальных страниц той или иной программы и определяет, находится ли она в оперативной памяти, и если находится, то в каком конкретно месте. Прикладные программы не касаются процесса страничного преобразования адреса и могут использовать все адресное пространство. Процессор автоматически формирует особый случай не присутствия, когда программа обращается к странице, отсутствующей в физической памяти. При обработке этого особого случая ОС загружает затребованную страницу из внешней памяти, при необходимости отправляя некоторую другую страницу на диск (процесс свопинга).



Рисунок: Принцип преобразования виртуального страничного адреса в физический

БОЛЕЕ ПОДРОБНО: СТРАНИЦА 22 УЧЕБНИКА 11. Гуров В.В. Архитектура микропроцессоров [Электронный ресурс]/ Гуров В.В.— Электрон. текстовые данные.— М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 115 с.— Режим доступа: <http://www.iprbookshop.ru/56313>.— ЭБС «IPRbooks» **ТАКЖЕ ТАМ ВЫ СКОРЕЕ ВСЕГО НАЙДЕТЕ ОТВЕТЫ НА ВОПРОСЫ С 4 ПО +∞ (НА ДРУГИХ СТРАНИЦАХ)**

8. Понятие «процесса» и «потока». Основные операции выполняемые над процессами и потоками.

Процесс — одно из фундаментальных понятий в любой ОС. К сожалению, однозначного определения этого термина не существует до сих пор. В большинстве случаев достаточно считать, что процесс — это единица работы

системы, которая описывается в системе в виде специальной структуры, часто называемой дескриптором процесса, и которой распределяются системные ресурсы.

Программный код только тогда начнёт выполняться, когда для него операционной системой будет создан процесс. Создать процесс — это значит:

- создать информационные структуры, описывающие данный процесс, то есть его дескриптор и контекст;
- включить дескриптор нового процесса в очередь готовых процессов;
- загрузить кодовый сегмент процесса в оперативную память или в область свопинга.

Операции над процессами

Над процессами можно производить следующие операции:

1. Создание процесса — это переход из состояния рождения в состояние готовности
2. Уничтожение процесса — это переход из состояния выполнения в состояние смерти
3. Восстановление процесса — переход из состояния готовности в состояние выполнения
4. Изменение приоритета процесса — переход из выполнения в готовность
5. Блокирование процесса — переход в состояние ожидания из состояния выполнения
6. Пробуждение процесса — переход из состояния ожидания в состояние готовности
7. Запуск процесса (или его выбор) — переход из состояния готовности в состояние выполнения

Для создания процесса операционной системе нужно:

1. Присвоить процессу имя
2. Добавить информацию о процессе в список процессов
3. Определить приоритет процесса
4. Сформировать блок управления процессом
5. Предоставить процессу нужные ему ресурсы

Иерархия процессов

Процесс не может взяться из ниоткуда: его обязательно должен запустить какой-то процесс. Процесс, запущенный другим процессом, называется дочерним (*child*) процессом или потомком. Процесс, который запустил новый процесс называется родительским (*parent*), родителем или просто — предком.

У каждого процесса есть два атрибута — PID (Process ID) — идентификатор процесса и PPID (Parent Process ID) — идентификатор родительского процесса.

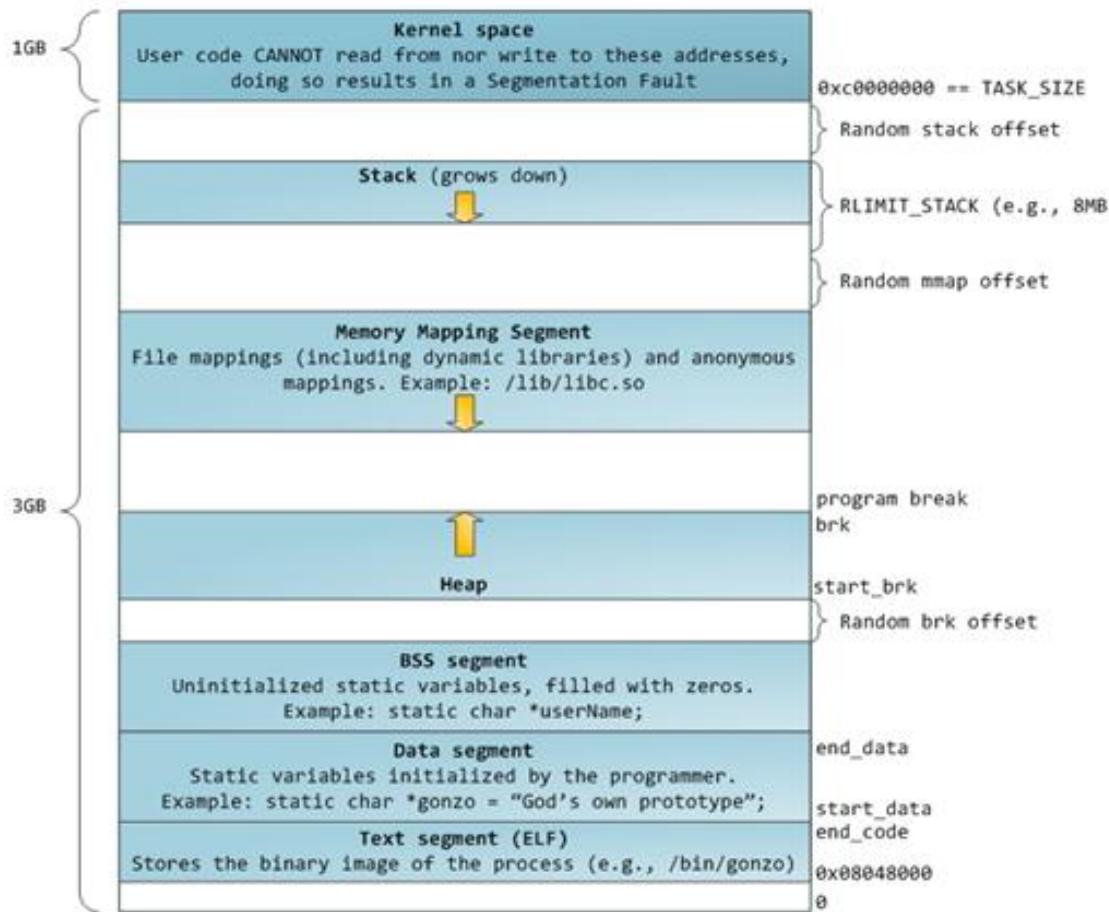
Процессы создают иерархию в виде дерева. Самым «главным» предком, то есть процессом, стоящим на вершине этого дерева, является процесс `init` (`PID=1`).

Поток — диспетчеризуемая единица работы, включающая контекст процессора (куда входит содержимое программного счётчика и указателя вершины стека), а также свою собственную область стека (для организации вызова подпрограмм и хранения локальных данных). Команды потока выполняются последовательно; поток может быть прерван при переключении процессора на обработку другого потока.

Важно понимать, что все потоки в рамках одного процесса используют общие ресурсы системы, выделенные данному процессу. Можно даже сказать, что процессы конкурируют за все ресурсы системы, кроме процессорного времени, в то время как потоки конкурируют только за процессорное время.

Концепция процесса, пришедшая из мира UNIX, плохо реализуется в многозадачной системе, поскольку процесс имеет тяжелый контекст. Возникает понятие потока (thread), который понимается как подпроцесс, или легковесный процесс (light-weight process), выполняющийся в контексте полноценного процесса.

9. Линейное адресное пространство процесса. Схема организации линейного адресного пространства процесса в ядре Linux.



Концепция виртуальной адресации распространяется на все выполняемое ПО, включая и само ядро. По этой причине для него резервируется часть виртуального адресного пространства (т.н. kernel space).

В верхней части user mode space расположен стековый сегмент. Большинство языков программирования используют его для хранения локальных переменных и аргументов, переданных в функцию.

Под стеком располагается сегмент для memory mapping. Ядро использует этот сегмент для мэппирования (отображения в память) содержимого файлов. Любое приложение может воспользоваться данным функционалом посредством системного вызова `malloc()`.

BSS и data сегмент хранят данные, соответствующий `static` переменным в исходном коде на С. Разница в том, что в BSS хранятся данные, соответствующие неинициализированным переменным, чьи значения явно не указаны в исходном коде.

Сегмент кода доступен только на чтение и хранит весь Ваш код и такие мелочи, как, например, строковые литералы.

10. Прерывания в архитектуре x86. Программируемые контроллер прерываний (PIC), улучшенный программируемый контроллер прерываний (APIC).

Прерывание определяется как событие, меняющее последовательность инструкций, выполняемых процессором. Синхронные прерывания выдаются управляющим блоком процессора при выполнении инструкций (исключения). Асинхронные генерируются другими аппаратными устройствами в произвольные моменты времени (прерывания).

Все IRQ-линии в системе соединены со входами электронной системы, которая называется программируемым контроллером прерываний, она выполняет следующие действия: **Ведёт мониторинг IRQ-линий**, проверяя наличие сигналов. Если возбуждены несколько линий, выбирает ту, у которой меньше номер входа. **Если на IRQ-линии возник сигнал**: преобразует принятый сигнал в соответствующий вектор; сохраняет вектор в порте ввода/вывода контроллера прерываний, тем самым позволяя процессору прочитать его через шину данных; посыпает сигнал на вход INTR процессора, т. е. возбуждает прерывания; ждет, пока процессор подтвердит прием сигнала прерывания, записав соответствующую информацию в один из портов ввода/вывода программируемого контроллера прерываний. Когда это произойдет, сбрасывает линию INTR. **Возвращается к шагу 1.**

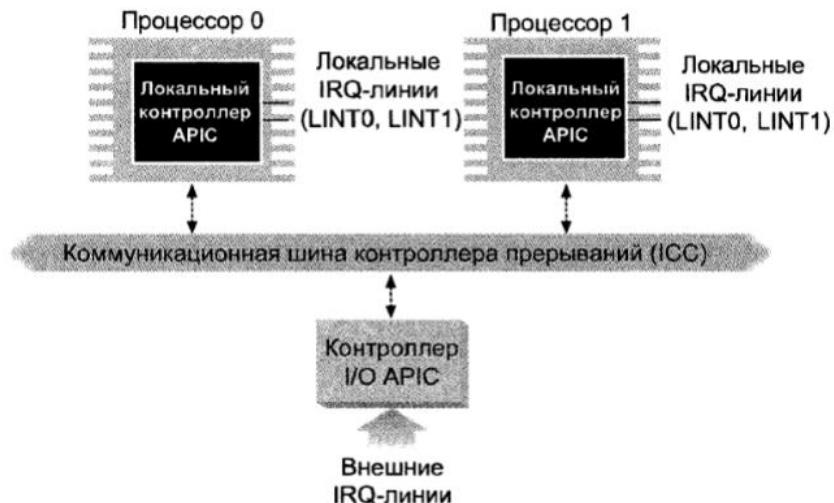


Рис. 4.1. Многоконтроллерная APIC-система

Улучшенный контроллер прерываний:

11. Обработка прерываний в архитектуре x86. Таблица дескрипторов прерываний IDT.

Обработка прерываний: Определяет вектор i ($0 < i < 255$), ассоциированный с этим прерыванием или исключением. Читает i -ю запись таблицы дескрипторов прерываний, на которую указывает регистр idtr. Получает базовый адрес таблицы GDT из регистра gdtr и читает в ней дескриптор сегмента, идентифицируемый селектором в записи таблицы дескрипторов прерываний. Этот дескриптор содержит базовый адрес сегмента, содержащего обработчик прерывания или исключения. Убеждается, что прерывание было возбуждено источником, имеющим на то право. Проверяет, имеет ли место изменение уровня привилегий. Если возникла ошибка, управляющий блок загружает в регистры cs и eip логический адрес инструкции, вызвавшей исключение, чтобы была возможность выполнить ее снова. Сохраняет в стеке содержимое регистров eflags, cs и eip. Если исключение несет в себе код аппаратной ошибки, управляющий блок сохраняет его в стеке. Загружает в регистры cs и eip, соответственно, значения селектора сегмента и смещения из дескриптора шлюза, хранящихся в i -й записи таблицы дескрипторов прерываний. Эти значения определяют логический адрес первой инструкции обработчика

Дескриптор шлюза задачи														
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32														
Зарезервировано					P	D	0	1	0					
Зарезервировано														
Селектор сегмента TSS					Зарезервировано									
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0														
Дескриптор шлюза прерывания														
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32					P	D	0	1	1					
Смещение (16-31)					0	1	1	1	0					
Зарезервировано					0	0	0	0	0					
Селектор сегмента TSS														
Смещение (0-15)														
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0														
Дескриптор шлюза ловушки														
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32					P	D	0	1	1					
Смещение (16-31)					1	1	1	1	0					
Зарезервировано					0	0	0	0	0					
Селектор сегмента TSS														
Смещение (0-15)														
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0														

Рис. 4.2. Формат дескрипторов шлюзов

прерывания или исключения.

В таблице содержатся следующие дескрипторы: **дескриптор шлюза задач** — включает в себя селектор TSS-сегмента процесса, который должен заместить текущий, когда возникнет сигнал прерывания; **дескриптор шлюза прерываний** — включает в себя селектор сегмента и смещение внутри сегмента обработчика прерывания или исключения. При передаче управления соответствующему сегменту процессор сбрасывает флаг if, тем самым отключая последующие маскируемые прерывания; **дескриптор шлюза ловушек** — аналогичен предыдущему, но при передаче управления нужному сегменту процессор не изменяет флаг if.

12. Режимы работы процессора с архитектурой x86. Привилегированный и не привилегированный режимы работы процессора.

По способу адресации памяти: Реальный режим: обращение к оперативной памяти происходит по реальным (действительным) адресам. Набор доступных операций не ограничен, защита памяти не используется. Защищённый режим: обращение к памяти происходит по виртуальным адресам с использованием механизмов защиты памяти. Набор доступных операций определяется уровнем привилегий.

По уровню привилегий: Режим пользователя (прикладной): минимальный уровень привилегий, разрешены только операции с данными и переходы в пределах адресного пространства пользователя. Все остальные операции либо игнорируются, либо с помощью механизма обработки исключений вызывают переключение в **привилегированный режим** и передачу управления ядру операционной системы для выполнения специальных функций (например, отображения данных на дисплее) или аварийного завершения потока управления. Привилегированный режим (режим ядра): наравне с операциями режима пользователя, разрешены дополнительные операции — запрет или разрешение прерываний, доступ к портам ввода-вывода, специальным регистрам процессора (например, для настройки блока управления памятью).

13. Системные вызовы. Способы выполнения системного вызова при помощи программного прерывания и ассемблерной инструкции sysenter

В вычислительной технике, **системный вызов** является программным способом, в котором компьютерная программа запрашивает определенную операцию от ядра операционной системы. Иными словами, системный вызов возникает, когда пользовательский процесс требует некоторой службы реализуемой ядром и вызывает специальную функцию.

Сюда могут входить услуги, связанные с аппаратным обеспечением (например, доступ к жесткому диску), создание и выполнение новых процессов, связь с интегральными службами ядра, такими как планирование процессов. Системные вызовы обеспечивают необходимый интерфейс между процессом и операционной системой

Программное прерывание — синхронное прерывание, которое может осуществить программа с помощью специальной инструкции.

Программное прерывание реализует один из способов перехода на подпрограмму с помощью специальной инструкции процессора, такой как INT в процессорах Intel

Практически все современные процессоры имеют в системе команд инструкции **программных прерываний**. Одной из причин появления инструкций программных прерываний в системе команд процессоров является то, что их использование часто приводит к более компактному коду программ по сравнению с использованием стандартных команд выполнения процедур. Это объясняется тем, что разработчики процессора обычно резервируют для обработки прерываний небольшое число возможных подпрограмм, так что длина операнда в команде программного прерывания, который указывает на нужную подпрограмму, меньше, чем в команде перехода на подпрограмму. Другой причиной применения программных прерываний вместо обычных инструкций вызова подпрограмм является возможность смены пользовательского режима на

привилегированный одновременно с вызовом процедуры — это свойство программных прерываний поддерживается большинством процессоров.

В результате программные прерывания часто используются для выполнения ограниченного количества вызовов функций ядра операционной системы, то есть **системных вызовов**.

Команда SYSENTER - это команда ассемблера, команда системного вызова, предназначена для быстрого перехода в режим ядра.

Режим ядра (англ. kernel mode) — привилегированный режим работы процессора, как правило, используемый для выполнения ядра операционной системы. В данном режиме работы процессора доступны привилегированные операции, такие, как операции ввода-вывода к периферийным устройствам, изменение параметров защиты памяти, настройка виртуальной памяти, системных параметров и прочих параметров конфигурации. Как правило, в режиме супервизора или вообще не действуют ограничения защиты памяти, или же они могут быть произвольным образом изменены, поэтому код, работающий в данном режиме, как правило, имеет полный доступ ко всем системным ресурсам (адресное пространство, регистры конфигурации процессора и так далее). Во многих типах процессоров это наиболее привилегированный режим из всех доступных режимов.

14. Мультипрограммный режим работы операционной системы. Системный планировщик процессов ядра Linux, полностью справедливый планировщик.

Мультипрограммный режим работы. Современные вычислительные системы функционируют, как правило, в мультипрограммном режиме, в котором выполняется несколько программ одновременно. Для вычислительных систем с несколькими процессорами число одновременно выполняемых программ обычно существенно превосходит число используемых процессоров.

В режиме мультипрограммирования в основной памяти одновременно находится несколько программ, загруженных для выполнения. Число одновременно выполняемых программ определяет уровень мультипрограммирования. Центральный процессор в каждый момент времени может выполнять лишь одну из программ. Таким образом, параллельно выполняемые программы конкурируют между собой за обладание ресурсами вычислительной системы и в первую очередь за время центрального процессора. Каждая программа представляется в системе как задача (процесс). Таким образом, принято говорить об одновременно выполняемых задачах (или процессах), основой которых являются соответствующие программы, причем задача является единицей

мультипрограммирования.

Системный планировщик Linux

➤ Компонент ядра, определяющий, какой из процессов должен выполняться, в какой именно момент времени и насколько долго

Версия ядра Linux	Тип планировщика
До версии 2.4	Простой планировщик, плохо поддающийся масштабированию
2.5	Планировщик типа $O(1)$
2.6	Планировщик типа RSDL (Rotating Staircase Deadline – Циклический ступенчатый граничный планировщик)
2.6.23	Планировщик типа CFS (Completely Fair Scheduler – Полностью справедливый планировщик)

Активация
Чтобы активировать планировщик CFS, выполните "Планировщик CFS"

Приоритет процессов

- Цель приоритетного планирования состоит в упорядочении процессов в соответствии с их важностью и необходимостью использования процессорного времени
- Процессы с более высоким приоритетом должны выполняться раньше тех, которые имеют более низкий приоритет
- Процессы с одинаковым приоритетом планируются к выполнению по циклическому алгоритму Round Robin

Динамическое назначение приоритетов

- Приоритет процессов, ограниченных скоростью ввода-вывода, в качестве награды, снижается на величину до пяти уровней
- Процессы, ограниченные производительностью ЦП, «штрафуются» повышением приоритета также на величину до пяти уровней
- Принадлежность процесса к классу ограниченных скоростью ввода-вывода или ограниченных производительностью ЦП определяется с помощью эвристической процедуры вычисления интерактивности
- Коррекция приоритета производится только для пользовательских процессов

Полностью справедливый планировщик

В планировщике задач CFS вместо очередей процессов ожидающих выполнения, используется дерево red-black-tree, определяющее план с временем перехода к выполнению очередного процесса. Единица планирования времени в CFS фиксирована - наносекунда, и не привязана к частоте генерации прерываний таймера (HZ).

15. Определение файловой системы. Примеры файловых систем, используемых в современных операционных системах

Файловая система - это часть операционной системы, назначение которой состоит в том, чтобы обеспечить пользователю удобный интерфейс при работе с данными, хранящимися на диске, и обеспечить совместное использование файлов несколькими пользователями и процессами.

В широком смысле понятие "файловая система" включает:

- совокупность всех файлов на диске,
- наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске,
- комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и другие операции над файлами.

Журналируемая файловая система — файловая система (ФС), в которой осуществляется ведение журнала, хранящего список изменений и, в той или иной степени, помогающего сохранить целостность файловой системы при сбоях.

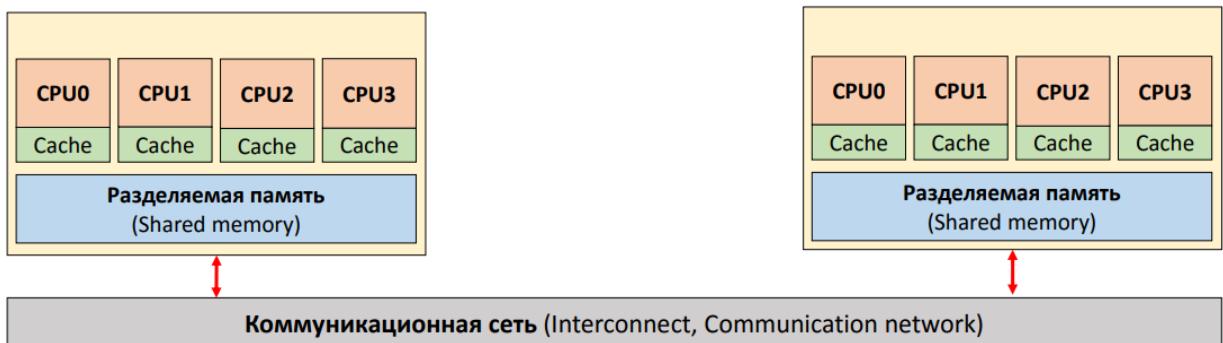
Виды:

- Ext3 - журналируемая файловая система, используемая в ОС на ядре Linux. Является файловой системой по умолчанию во многих дистрибутивах. Основана на Ext2, но отличается тем, что в ней есть журналирование, то есть в ней предусмотрена запись некоторых данных, позволяющих восстановить файловую систему при сбоях в работе компьютера.
- Ext2 - файловая система, используемая в операционных системах на ядре Linux. Достаточно быстра для того, чтобы служить эталоном в тестах производительности файловых систем. Она не является журналируемой файловой системой и это её главный недостаток
- Ext4 - журналируемая файловая система, используемая в ОС на ядре Linux. Основана на файловой системе Ext3, но отличается тем, что в ней представлен механизм записи файлов в непрерывные участки блоков (екстенты), уменьшающий фрагментацию и повышающий производительность.
- XFS - высокопроизводительная журналируемая файловая система. Распределение дискового пространства - екстентами, хранение каталогов в В-деревьях. Автоматическая аллокация и вы свобождение I-node. Дефрагментируется «на лету». Невозможно уменьшить размер существующей файловой системы. При сбое питания во время записи возможна потеря данных (хотя этот недостаток нельзя относить к одной только XFS, он свойственен любой журналируемой ФС, но, вместе с тем, XFS, по умолчанию, достаточно активно использует буферы в памяти).
- Fat16 - файловая система, сейчас широко используемая в картах памяти фотоаппаратов и других устройств.

- Fat32 - файловая система основанная на Fat16. Создана, чтобы преодолеть ограничения на размер тома в Fat16.
- NTFS - файловая система для семейства операционных систем Microsoft Windows.
- HFS - файловая система, разработанная Apple Inc. для использования на компьютерах, работающих под управлением операционной системы Mac OS.

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ ТЕХНОЛОГИИ

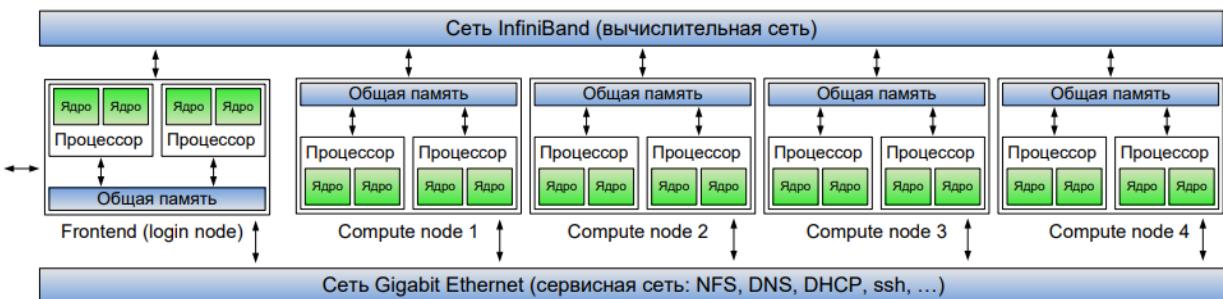
1. Архитектура вычислительных систем с распределенной памятью, конфигурация вычислительных узлов, структуры коммуникационных сетей. Гибридные вычислительные системы на базе специализированных ускорителей.



Вычислительная система с распределенной памятью (Distributed memory computer system) – совокупность вычислительных узлов, взаимодействие между которыми осуществляется через коммуникационную сеть (InfiniBand, Gigabit Ethernet, Cray Gemeni, Fujitsu Tofu, ...). Каждый узел имеет множество процессоров/ядер, взаимодействующих через разделяемую память (Shared memory).

Процессоры могут быть многоядерными, ядра могут поддерживать одновременную многопоточность (SMT).

Отдельным примером такой вс являются кластеры.



Основные особенности:

Вычислительные кластеры строятся на базе свободно доступных компонентов

Вычислительные узлы: 2/4-процессорные узлы, 1 – 8 GiB оперативной памяти на ядро (поток)

Коммуникационная сеть (сервисная и для обмена сообщениями)

Подсистема хранения данных (дисковый массивы, параллельные и сетевые файловые системы)

Система бесперебойного электропитания

Система охлаждения

Программное обеспечение: GNU/Linux (NFS, NIS, DNS, ...), MPI (MPICH2, Open MPI), TORQUE/SLURM

Плюсы

Высокая масштабируемость (сотни, тысячи и миллионы процессорных ядер)

Меньше проблем с синхронизацией (у каждого узла/сервера своя память)

Декомпозиция на крупные подзадачи

Минусы

Необходимость использования передачи сообщений (message passing)

Высокие временные задержки и низкая пропускная способность (все взаимодействия по сети – Gigabit Ethernet, Infiniband)

Неоднородность, отказы узлов

Коммуникационные сети.

Задачи коммуникационных сетей ВС (communication network, interconnect):

Реализация обменов информацией между ветвями параллельных программ:

односторонние обмены (one-sided, RDMA: put/get), двусторонние (индивидуальные, дифференцированные, point-to-point: send/recv), коллективные операции (collectives: one-to-all broadcast, all-to-one gather/reduce, all-to-all)

Реализация обменов служебной информацией: контроль и диагностика состояния вычислительных узлов, барьерная синхронизация

Функционирования сетевых и параллельных файловых систем (доступ к дисковым массивам)

Требования к коммуникационной сети:

Высокая производительность реализации всех видов обменов (двусторонних, коллективных) – адекватность структуры ВС широкому классу параллельных алгоритмов

Масштабируемость (простое увеличение и уменьшение числа ЭМ в системе)

Живучесть и отказоустойчивость (функционирование при отказах отдельных подсистем)

Высокая технико-экономическая эффективность (цена/эффективность)

Виды коммуникационных сетей:

1. С фиксированной структурой (статической топологией) – графиком связей вычислительных узлов, при этом каждый вычислительный узел имеет сетевой интерфейс (маршрутизатор) с несколькими портами, через который он напрямую соединён с другими узлами
2. С динамической структурой – на базе коммутаторов, при этом каждый вычислительный узел имеет сетевой интерфейс с несколькими портами. Порты интерфейсов подключены к коммутаторам, через которые происходит взаимодействие узлов

Сети с фиксированной структурой:

Структура сети (топология) – это график, где

узлы – машины (вычислительные узлы, computer nodes)

ребра – линии связи (links)

Показатели эффективности структур:

диаметр (максимальное из кратчайших расстояний)

средний диаметр сети

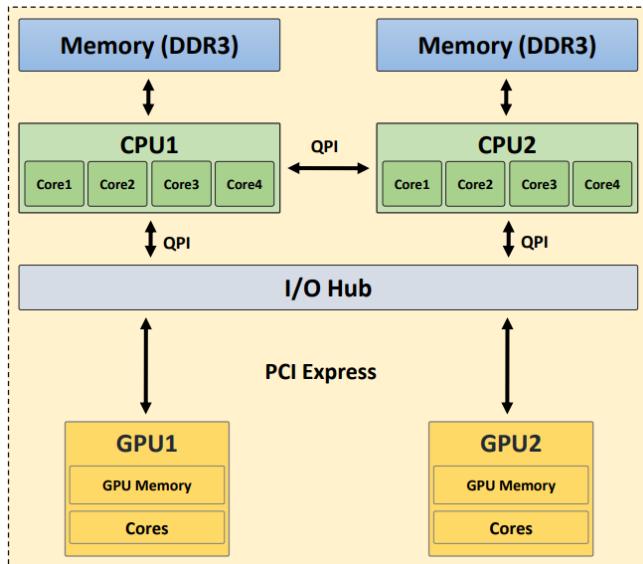
бисекционная пропускная способность (bisectional bandwidth)

Примеры структур

kD-тор (3D, 4D, 5D): Cray Titan 3D torus, IBM Sequoia 5D torus, Fujitsu 6D torus

Гиперкуб, решетка, кольцо, графы Кауца, циркулянтные структуры, ...

Гибридные вычислительные системы на базе специализированных ускорителей.



До последнего времени ключевым компонентом систем для высокопроизводительных вычислений, включая кластеры, был центральный процессор. Однако несколько лет назад у него появился серьезный конкурент – графический процессор (GPU).

Высокая производительность GPU объясняется особенностями его архитектуры.

В отличие от центрального процессора, который состоит из нескольких ядер, графический процессор изначально создавался как многоядерная структура, в которой количество компонентов измеряется сотнями. Например, в графическом процессоре NVIDIA поколения «Fermi» 512 вычислительных ядер. Также есть существенная разница и в принципах работы – архитектура CPU предполагает последовательную обработку информации, а GPU исторически предназначался для обработки компьютерной графики, поэтому рассчитан на массивно параллельные вычисления. Каждая из этих двух архитектур имеет свои достоинства. Говорить об абсолютной замене CPU на GPU не имеет смысла – они не взаимозаменяют, а дополняют друг друга. CPU лучше работает с последовательными задачами, но при большом объеме обрабатываемой информации, с которой можно работать параллельно, очевидное преимущество имеет GPU.

Пиковая производительность современного GPU в разы выше, чем производительность современного CPU. В скорости доступа к видеопамяти GPU также имеет значительное превосходство над CPU. Эффективная организация подсистемы памяти повышает общую эффективность графического процессора при работе с неграфическими задачами.

Высокая производительность массивно-параллельной архитектуры GPU обеспечивает гибридные системы колоссальной производительностью при решении самого широкого класса научных и прикладных вычислительных задач.

Стоит отметить, что для работы с GPU используется программно-аппаратная платформа

CUDA, позволяющая разрабатывать код, обрабатывающийся непосредственно на графических ядрах.

2. Показатели эффективности параллельных алгоритмов и программ: коэффициент ускорения, коэффициент накладных расходов. Анализ строгой и слабой масштабируемость параллельных программ.

Коэффициент ускорения $S_p(n)$ показывает во сколько раз параллельная программа выполняется на p процессорах быстрее последовательной программы при обработке одних и тех же входных данных размера n .

$$S_p(n) = \frac{T(n)}{T_p(n)},$$

где

$T(n)$ — время выполнения последовательной программы(sequential program),

$T_p(n)$ — время выполнения параллельной программы (parallel program) на p процессорах

Цель распараллеливания – достичь линейного ускорения на максимально большом числе процессоров

$$S_p(n) = \Omega(p) \text{ при } p \rightarrow \infty$$

Коэффициент относительного ускорения (Relative speedup) – отношения времени выполнения параллельной программы на k процессорах к времени её выполнения на p процессорах ($k < p$).

$$S_{Relative}(k, p, n) = \frac{T_k(n)}{T_p(n)},$$

Коэффициент эффективности (Efficiency) параллельной программы.

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T(n)}{pT_p(n)} \in [0,1],$$

Коэффициент накладных расходов (Overhead).

$$\varepsilon(p, n) = \frac{T_{Sync}(n)}{T_{Comp}(n)} = \frac{T_{Total}(p, n) - T_{comp}(p, n)}{T_{comp}(p, n)},$$

Масштабируемость параллельной программы (scalability) – характеристика программы, показывающая как изменяются ее показатели производительности при варьировании числа параллельных процессов на конкретной ВС.

Строгая/сильная масштабируемость (strong scaling) – зависимость коэффициента ускорения от числа p процессов при фиксированном размере n входных данных ($n = \text{const}$)

Показывает как растут накладные расходы с увеличением p .

Цель – минимизировать время решения задачи фиксированного размера.

Слабая масштабируемость (weak scaling) – зависимость коэффициента ускорения параллельной программы от числа процессов при фиксированном размере входных данных на один процессор ($n / p = \text{const}$)

Цель – решить задачу наибольшего размера на ВС.

Параллельная программа (алгоритм) коэффициент ускорения, которой линейной растет с увеличением p называется линейно масштабируемой или просто масштабируемой (scalable).

3. Понятие масштабируемых программ. Законы Амдала и Густафсона-Барсиса.

Закон Дж. Амдала:

Иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей.

Пусть имеется последовательная программа с временем выполнения $T(n)$.

Обозначим:

- $r \in [0, 1]$ — часть программы, которая может быть распараллелена,
- $s = 1 - r$ — часть программы, которая не может быть распараллелена.

Максимальное ускорение S_p программы на p процессорах равняется

$$S_p = \frac{1}{(1-r) + \frac{r}{p}}$$
$$S_\infty = \lim_{p \rightarrow \infty} S_p = \lim_{p \rightarrow \infty} \frac{1}{(1-r) + \frac{r}{p}} = \frac{1}{1-r} = \frac{1}{s}$$

Допущения закона Дж. Амдала:

1. Последовательный алгоритм является наиболее оптимальным способом решения задачи
 - a. Возможны ситуации когда параллельная программа (алгоритм) эффективнее решает задачу (может эффективнее использовать кеш-память, конвейер, SIMD-инструкции, ...)
2. Время выполнения параллельной программы оценивается через время выполнения последовательной, однако потоки параллельной программы могут выполняться эффективнее
3. Ускорение $S_p(n)$ оценивается для фиксированного размера n данных при любых значениях p
 - a. В реальности при увеличении числа используемых процессоров размер n входных данных также увеличиваются, так как может быть доступно больше памяти

Закон Густафсона-Барсиса:

Оценка максимально достижимого ускорения выполнения параллельной программы, в зависимости от количества одновременно выполняемых потоков вычислений («процессоров») и доли последовательных расчётов.

Пусть имеется последовательная программа с временем выполнения $T(n)$.

Обозначим:

- $s \in [0, 1]$ — часть параллельной программы, которая выполняется последовательно

Масштабируемое ускорение S_p программы на p процессорах равняется

$$S_p = p - s(p - 1)$$

Обоснование: пусть a — время последовательной части, b — время параллельной части

$$T_p(n) = a + b, T(n) = a + pb$$

$$s = \frac{a}{a + b}, S_p(n) = s + p(1 - s) = p - s(p - 1),$$

Время выполнения последовательной программы выражается через время выполнения параллельной.

Отличие от закона Амдала:

При оценке ускорения параллельного выполнения закон Амдала предполагает, что объем задачи остается постоянным. Величина ускорения по закону Амдала показывает, во сколько раз меньше времени потребуется параллельной программе для выполнения.

Однако величину ускорения можно рассматривать и как увеличение объема выполненной задачи за постоянный промежуток времени. Закон Густафсона появился именно из этого предположения.

4. Основные понятия многопоточного программирования: взаимные блокировки и «гонка данных». Синхронизация: мьютексы и семафоры.

Самые первые операционные системы могли выполнять только одну программу одновременно. Все ресурсы системы были доступны для одной программы. Вскоре, операционные системы могли выполнять несколько программ одновременно, их чередованием. Программы должны были заранее указать, какие ресурсы им необходимы, чтобы они могли избежать конфликтов с другими программами, работающими одновременно. В конце концов некоторые операционные системы предложили динамическое распределение ресурсов. Программы могли запрашивать дополнительные ресурсы после того, как они начали работать. Это привело к проблеме взаимной блокировки.

Взаимная блокировка (deadlock) – ситуация, когда один и более потоков находятся в состоянии бесконечного ожидания ресурсов, в результате чего программа способна прекратить функциональность. Ресурсы могут быть как физическими, так и логическими. Примерами логических ресурсов являются файлы, семафоры и мониторы. К физическим можно отнести принтеры, память, такты процессора.

Одним из наиболее распространённых факторов появления данной проблемы является не правильное использование мьютексов.

Состояние гонки (Race condition, data race) – это состояние программы, в которой несколько потоков одновременно конкурируют за доступ к общей структуре данных (для чтения/записи).

Например. Пусть два параллельно работающих потока выполняют некоторый фрагмент кода. Первый поток выполняет оператор присваивания:

Sum += v1;

Второй поток выполняет оператор присваивания:

Sum += v2;

Ожидаемым результатом должно быть увеличение суммы Sum, как на величину первой, так и второй переменной. В ряде случаев из-за возникшей "гонки данных" сумма увеличится только на величину одной переменной, и нельзя сказать какой именно.

Проблема в том, что оба оператора присваивания работают с общим ресурсом — переменной Sum. Оператор присваивания, рассматриваемый на уровне языка программирования, как одна операция, на уровне компьютера после трансляции превратится в группу команд.

Оба потока могут одновременно прочесть из памяти, отводимой переменной Sum, ее текущее значение и занести его на соответствующие регистры. Затем одновременно выполнить сложение в регистровой памяти. Но записать полученный результат в

ячейку Sum придется последовательно. Тот поток, кто пришел первым в гонке данных, проиграет, его результат будет утерян, после того, как в эту же ячейку запишет результат второй поток. Конечно же, если эти потоки команды будут смешены по времени, и одна группа команд будет выполняться по завершении работы первой группы, то в сумме будут учтены оба вклада. Но чтобы добиться такого результата, необходимо предпринять определенные меры по синхронизации работы параллельно работающих потоков.

Семафор — это примитив синхронизации, позволяющий ограничить доступ к критической секции только для N потоков. При этом, как правило, семафор позволяет реализовать это без использования занятого ожидания. Концептуально семафор включает в себя счетчик и очередь ожидания для потоков. Интерфейс семафора состоит из двух основных операций: опустить (down), поднять (up). Операция опустить атомарно проверяет, что счетчик больше 0 и уменьшает его. Если счетчик равен 0, поток блокируется и становится в очередь ожидания. Операция поднять увеличивает счетчик и посыпает ожидающим потокам сигнал пробудиться, после чего один из этих потоков сможет повторить операцию опустить.

Бинарный семафор — это семафор с N = 1.

Мьютекс — от словосочетания mutual exclusion, т.е. взаимное исключение — это примитив синхронизации, напоминающий бинарный семафор с дополнительным условием: разблокировать его должен тот же поток, который и заблокировал.

Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищенным мьютексом. Если другому потоку будет нужен доступ к переменной, защищенной мьютексом, то этот поток блокируется до тех пор, пока мьютекс не будет освобождён.

Цель использования мьютексов — защита данных от повреждения в результате асинхронных изменений (состояние гонки), однако могут порождаться другие проблемы — например взаимная блокировка (клинич).

5-7(Основные понятия многопоточного программирования)

Поток – последовательность команд программы, выполняющихся одна за другой в детерминированной последовательности.

Основной принцип потока – несколько потоков могут выполняться параллельно (пseudопараллельно).

Поток (поток управления, задача, нить, thread) – одна из параллельно (асинхронно) выполняющихся ветвей процесса

- Особенности потоков
 - В процессе присутствует единственный главный поток
 - Все потоки одного процесса работают в едином адресном пространстве
 - Общие переменные и код
 - Нет необходимости использовать специальные средства взаимодействия*
 - Каждый поток имеет собственный стек
 - Каждый поток имеет собственное состояние
- межпоточного взаимодействия обычно нужны

Процесс – выполнение пассивных инструкций компьютерной программы на процессоре.

Процесс – совокупность взаимосвязанных и взаимодействующих действий, преобразующих входные данные в выходные (ISO 9000:2000).

Процесс порождается при запуске программы.

Состав процесса:

- Главный поток
- Дополнительные потоки – необязательно
- Память данных
- Память программ

Все потоки в рамках одного процесса выполняются в едином адресном пространстве, то есть используют общие переменные. Каждый поток имеет собственный стек. Поток может порождать другие потоки.

Процесс (process) – совокупность действий процессора и необходимых ресурсов для обеспечения выполнения инструкций программы

■ Состав процесса

- Области памяти данных и программ
- Стек
- Отображение виртуальной памяти на физическую память
- Состояние

Типичные состояния процессов:

- Остановлен
 - Процесс не использует процессор
- Терминирован
 - Процесс закончился, но ещё не удалён
- Ожидает
 - Процесс ждёт событие
- Готов
 - Готов к выполнению, но ожидает освобождения процессора
- Выполняется
 - Процесс выполняется процессором

На рисунке 8 представлен график состояний процесса.

Межпроцессное взаимодействие – способ передачи информации между процессами.

- Виды межпроцессного взаимодействия
 - Разделяемая память
 - Семафоры
 - Сигналы
 - Почтовые ящики

Событие – оповещение процесса со стороны ОС о возникновении межпроцессного взаимодействия.

Примеры событий:

- Принятие семафором требуемого значения
- Поступление сигнала
- Поступление сообщения в почтовый ящик

Ресурс – объект (устройство, память), необходимый процессу или потоку для выполнения заданных действий

Приоритет – целое число, определяющее важность каждого процесса или потока в ОС:

- Чем больше приоритет у процесса или потока, тем больше процессорного времени ему будет выделено

Критическая секция – участок программного кода, который допускается выполнять только единственным потоком (процессом)

Взаимное исключение (мьютекс, mutual exclusion, mutex) – способ синхронизации потоков за счёт использования захвата совместно используемого ресурса, также называемого мьютексом

- Если мьютекс занят, то при попытке его захвата поток переходит в состояние ожидания
- Как только мьютекс освобождается, ранее ожидавший поток высвобождается, а мьютекс вновь считается захваченным

Безопасное взаимодействие – целостность информации и неделимость действий при взаимодействии обеспечиваются операционной системой

Небезопасное взаимодействие – целостность информации и неделимость действий при взаимодействии обеспечиваются приложением

Разделяемая память – область памяти, одновременно доступная для нескольких процессов.

Семафоры - это объект синхронизации, задающий количество процессов и/или потоков, имеющих одновременный доступ к разделяемому ресурсу

Многозадачность (multitasking) – свойство ОС или среды исполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких процессов.

5(Атомарные операции)

Атомарные операции — операции, выполняющиеся как единое целое либо не выполняющиеся вовсе.

6(Операции редукции)

Операции редукции объединяют элементы входных данных каждого процесса и возвращают объединенное значение в процесс с номером root. Пример: максимальное значение переменной среди всех процессов группы.

7(Потокобезопасные структуры данных)

Потокобезопасные структуры данных необходимы для исключения одновременного взаимодействия процессов(потоков) с разделяемыми данными. Пример: чтение данных одним процессом, во время изменения этих же данных другим.

Очереди необходимы для последовательного обращения к данным разными процессами. Пример: пока один поток изменяет разделяемые данные, другие потоки ждут своей очереди на обращение к этим данным.

*Одновременное чтение данных разными процессами (при условии что ни один процесс их не изменяет) считается потокобезопасным.

8(Модель передачи сообщений: стандарт MPI и его реализации.

Нумерация процессов и понятие коммуникатора.)

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.

- MPICH — одна из самых первых свободная реализация MPI, работает на UNIX-системах и Windows NT
- Open MPI — ещё одна свободная реализация MPI. Основана на более ранних проектах FT-MPI, LA-MPI, LAM/MPI и PACX-MPI.
Поддерживаются различные коммуникационные системы (в том числе Myrinet).
- WMPI — реализация MPI для Windows
- MPI/PRO for Windows NT — коммерческая реализация для Windows NT
- Intel MPI — коммерческая реализация для Windows / Linux

В MPI каждый процесс имеет свой уникальный номер. В процессе выполнения программы процесс может самостоятельно определить, присвоенный ему, номер, и выполнять те действия и вычисления, которые предназначены для этого номера.

Коммуникаторы бывают двух типов:

1. *интракоммуникаторы* - для операций внутри одной группы процессов;
2. *интеркоммуникаторы* - для двухточечного обмена между двумя группами процессов.

Интракоммуникатором является **MPI_COMM_WORLD**.

В MPI-программах чаще используются интракоммуникаторы.

Интракоммуникатор включает экземпляр группы, контекст обмена для всех его видов, а также, возможно, виртуальную топологию и другие атрибуты. Контекст обеспечивает возможность создания изолированных друг от друга, а потому безопасных областей взаимодействия. Система сама управляет их разделением. Контекст играет роль дополнительного тега, который дифференцирует сообщения.

9.Модель передачи сообщений: стандарт MPI и его реализации.

Двусторонние обмены стандарта MPI.

Двусторонний обмен (Point-to-point communication) – это обмен сообщением между двумя процессами: один инициирует передачу (**send**), а другой – получение сообщения (**receive**). Каждое сообщение снабжается конвертом (**envelope**):

- номер процесса отправителя (source)
- номер процесса получателя (destination)
- тег сообщения (tag)
- коммуникатор (communicator)

Блокирующие (Blocking)

- MPI_Bsend
- MPI_Recv
- MPI_Rsend
- MPI_Send
- MPI_Sendrecv
- MPI_Sendrecv_replace
- MPI_Ssend
- ...

Неблокирующие (Non-blocking)

- MPI_Ibsend
- MPI_Irecv
- MPI_Irsend
- MPI_Isend
- MPI_Issend
- ...

Пример Send

```
int MPI_Send(void *buf,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm)
```

- buf – указатель на буфер с информацией
- count – количество передаваемых элементов типа datatype
- dest – номер процесса получателя сообщения
- tag – тег сообщения

Пример Recv

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

Основное было выше, дальше идёт не совсем обязательная инфа, но также имеющая отношение к вопросу.

Режимы передачи сообщений

Standard (MPI_Send) – библиотека сама принимает решение буферизовать сообщение и сразу выходить из функции или дожидаться окончания передачи данных получателю (например, если нет свободного буфера)

Buffered (MPI_Bsend) – отправляемое сообщение помещается в буфер и функция завершает свою работу. Сообщение будет отправлено библиотекой.

Synchronous (MPI_Ssend) – функция завершает своё выполнение когда процесс-получатель вызвал MPI_Recv и начал копировать сообщение. После вызова функции можно безопасно использовать буфер.

Ready (MPI_Rsend) – функция успешно выполняется только если процесс-получатель уже вызвал функцию MPI_Recv

Неблокирующая передача/прием сообщения (Nonblocking point-to-point communication) – операция, выход из которой осуществляется не дожидаясь завершения передачи информации Пользователю возвращается дескриптор запроса, который он может использовать для проверки состояния операции Цель – обеспечить возможность совмещения вычислений и обменов информацией

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

```
int MPI_Wait(MPI_Request *request,
             MPI_Status *status)
```

Блокирует выполнение процесса пока не закончится операция, ассоциированная с запросом request

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

Возвращает flag = 1 если операция, ассоциированная с запросом request, завершена

10.Модель передачи сообщений: стандарт MPI и его реализации. Коллективные операции обмена информацией.

Трансляционный обмен (One-to-all)

- **MPI_Bcast** (рассылка информации от одного процесса всем остальным членам некоторой области связи)
- **MPI_Scatter** (разбиение массива и рассылка его фрагментов (**scatter**) всем процессам области связи)
- **MPI_Scatterv** (векторный вариант)

Коллекторный обмен (All-to-one)

- **MPI_Gather** (сборка (**gather**) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса)
- **MPI_Gatherv** (векторный вариант)
- **MPI_Reduce** (сохранение результата глобальной вычислительной операции в адресном пространстве одного процесса)

Трансляционно-циклический обмен (All-to-all)

- **MPI_Allgather** (сборка (**gather**) распределенного массива в один массив с рассылкой его всем процессам некоторой области связи)
- **MPI_Allgatherv**
- **MPI_Alltoall** (совмещенная операция **Scatter/Gather**, каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами в свой буфер приема)
- **MPI_Alltoallv**
- **MPI_Allreduce** (рассылка результата глобальной вычислительной операции всем процессам)
- **MPI_Reduce_scatter** (совмещенная операция **Reduce/Scatter**)

Главное отличие коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Все коммуникационные подпрограммы, за исключением **MPI_Bcast**, представлены в двух вариантах:

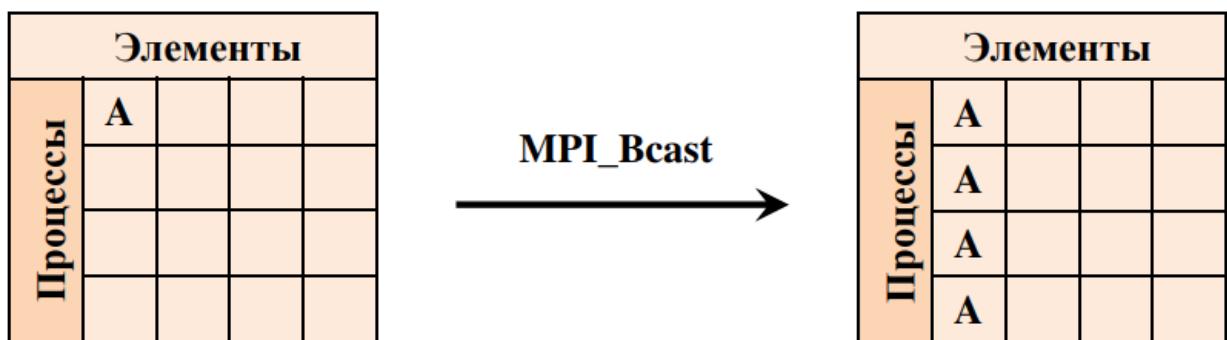
- простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;
- "векторный" вариант, который предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом "v" в конце имени функции.

Отличительные особенности коллективных операций:

- Коллективные коммуникации не взаимодействуют с коммуникациями типа точка-точка.
- Коллективные коммуникации выполняются в режиме с блокировкой. Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию.
- Количество получаемых данных должно быть равно количеству посланных данных.
- Типы элементов посылаемых и получаемых сообщений должны совпадать.
- Сообщения не имеют идентификаторов.

Далее несколько иллюстраций операций коллективных обменов.

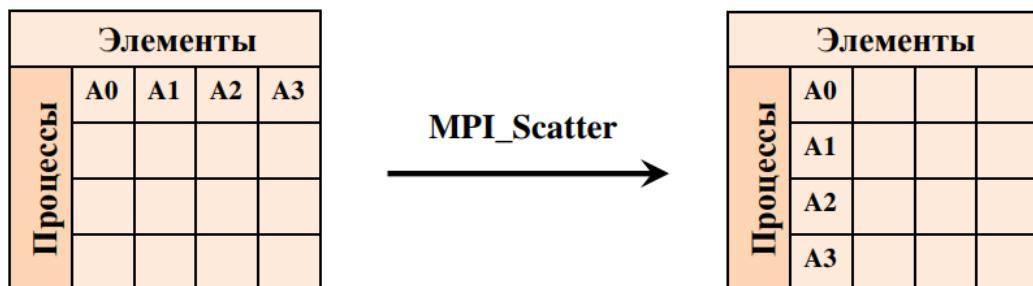
```
int MPI_Bcast(void *buf, int count,
              MPI_Datatype datatype,
              int root, MPI_Comm comm)
```



```

int MPI_Scatter(void *sendbuf, int sendcnt,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcnt,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)

```



```

int MPI_Gather(void *sendbuf, int sendcnt,
               MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype,
               int root, MPI_Comm comm)

```

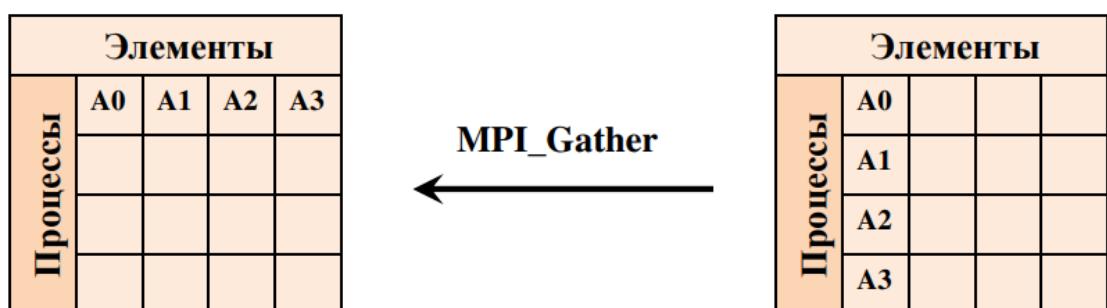
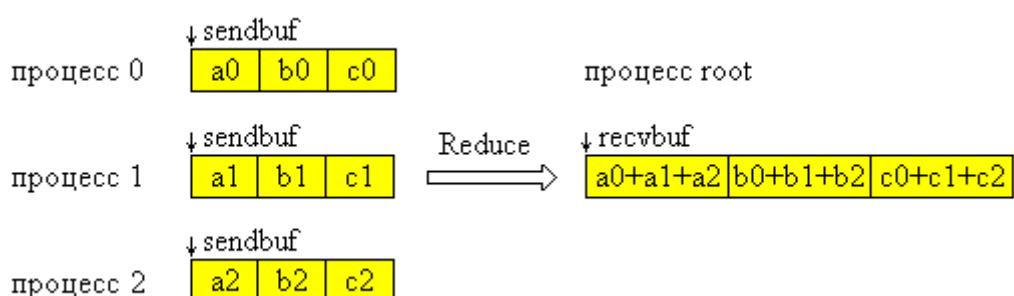


Иллюстрация операции Reduce на примере операции «+»



11.Модель передачи сообщений: стандарт MPI и его реализации. Производные типы данных.

Производные типы данных – это типы данных, которые построены из основных типов данных MPI

Производные типы MPI не являются в полном смысле типами данных, как это понимается в языках программирования. Они не могут использоваться ни в каких других операциях, кроме коммуникационных. Производные типы MPI следует понимать как описатели расположения в памяти элементов базовых типов. Производный тип MPI представляет собой скрытый (*opaque*) объект, который специфицирует две вещи: последовательность базовых типов и последовательность смещений. Последовательность таких пар определяется как *отображение (карта) типа*:

$$\text{Турепар} = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$$

Значения смещений не обязательно должны быть неотрицательными, различными и упорядоченными по возрастанию. Отображение типа вместе с базовым адресом начала расположения данных buf определяет коммуникационный буфер обмена. Этот буфер будет содержать n элементов, а i -й элемент будет иметь адрес $buf + disp$ и иметь базовый тип type. Стандартные типы MPI имеют предопределенные отображения типов. Например, MPI_INT имеет отображение $\{(int, 0)\}$.

Использование производного типа в функциях обмена сообщениями можно рассматривать как трафарет, наложенный на область памяти, которая содержит передаваемое или принятое сообщение.

Стандартный сценарий определения и использования производных типов включает следующие шаги:

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct.
- Новый производный тип регистрируется вызовом функции MPI_Type_commit. Только после регистрации новый производный тип можно использовать в коммуникационных подпрограммах и при конструировании других типов. Предопределенные типы MPI считаются зарегистрированными.
- Когда производный тип становится ненужным, он уничтожается функцией MPI_Type_free.

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах:

- *Протяженность типа* определяет, сколько байт переменная данного типа занимает в памяти. Эта величина может быть вычислена как: адрес последней ячейки данных - адрес первой ячейки данных + длина последней ячейки данных (опрашивается подпрограммой MPI_Type_extent).

- *Размер типа* определяет количество реально передаваемых байт в коммуникационных операциях. Эта величина равна сумме длин всех базовых элементов определяемого типа (опрашивается подпрограммой MPI_Type_size).

Для простых типов протяженность и размер совпадают.

Contiguous. Простейшим типом конструктора типа данных является конструктор MPI_TYPE_CONTIGUOUS, который позволяет копировать тип данных в смежные области.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

Новый тип newtype есть тип, полученный конкатенацией (сцеплением) count копий старого типа oldtype.

Vector. Функция MPI_TYPE_VECTOR является более универсальным конструктором, который позволяет реплицировать типы данных в области, состоящие из блоков равного объема. Каждый блок получается конкатенацией некоторого количества копий старого типа. Пространство между блоками кратно размеру old datatype.

Синтаксис функции MPI_TYPE_VECTOR представлен ниже.

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

Hvector. Функция MPI_TYPE_HVECTOR идентична MPI_TYPE_VECTOR за исключением того, что страйд задается в байтах, а не в элементах. (H обозначает heterogeneous - неоднородный).

Синтаксис функции MPI_TYPE_HVECTOR представлен ниже.

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)

Indexed. Функция MPI_TYPE_INDEXED позволяет реплицировать старый тип old datatype в последовательность блоков (каждый блок есть конкатенация old datatype), где каждый блок может содержать различное число копий и иметь различное смещение. Все смещения блоков кратны длине старого блока old type.

Синтаксис функции MPI_TYPE_INDEXED представлен ниже.

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

Hindexed. Функция MPI_TYPE_HINDEXED идентична MPI_TYPE_INDEXED, за исключением того, что смещения блоков в массиве array_of_displacements задаются в байтах, а не в кратностях величины старого типа oldtype.

Синтаксис функции MPI_TYPE_HINDEXED представлен ниже.

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)

Struct. MPI_TYPE_STRUCT является наиболее общим типом конструктора. Он отличается от предыдущего тем, что позволяет каждому блоку состоять из репликаций различного типа.

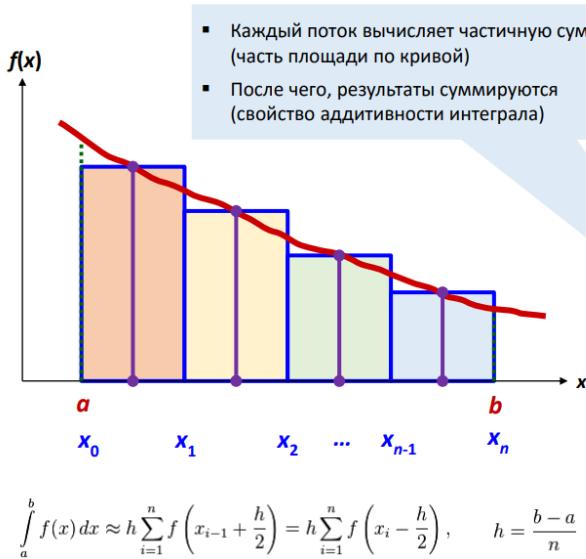
Синтаксис функции MPI_TYPE_STRUCT представлен ниже.

MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

12. Модель передачи сообщений. Подходы к распараллеливанию алгоритмов численного интегрирования: метод средних прямоугольников, метод Монте-Карло.

МЕТОД СРЕДНИХ ПРЯМОУГОЛЬНИКОВ

Параллельный алгоритм интегрирования методом средних прямоугольников (midpoint rule)



```

double func(double x)
{
    return exp(-x * x);
}

int main(int argc, char **argv)
{
    const double a = -4.0;
    const double b = 4.0;
    const int n = 100;

    double h = (b - a) / n;
    double s = 0.0;
    for (int i = 0; i < n; i++)
        s += func(a + h * (i + 0.5));
    s *= h;

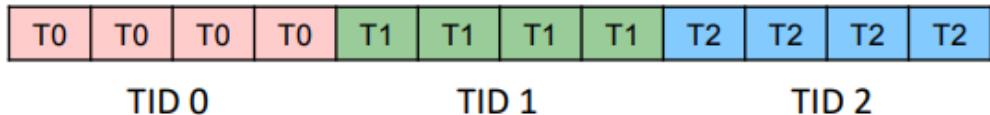
    printf("Result Pi: %.12f\n", s * s);
    return 0;
}

```

1. Итерации цикла *for* распределяются между потоками
2. Каждый поток вычисляет часть суммы (площади)
3. Суммирование результатов потоков (во всех или одном)

Варианты распределения итераций (точек) между потоками:

- 1) Разбиение на p смежных непрерывных частей



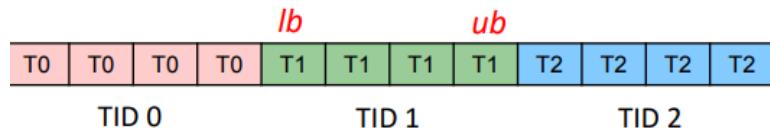
- 2) Циклическое распределение итераций по потокам



TID 0, TID 1, TID 2, TID 0, TID 1, TID 2, ...

Каждому потоку необходимо передать:

- порядковый номер потока
- число потоков
- h, n, a



При обновлении глобальной суммы необходимо воткнуть критических секций, чтобы избежать датарейса и данные не перетирались. (мютекс)

pdfина курносыча

МОНТЕ КАРЛО

Метод статистических испытаний (метод Монте-Карло) – численный метод решения математических задач, при котором:

- искомые величины представляют вероятностными характеристиками какого-либо случайного явления,
- это явление моделируется,
- нужные характеристики приближённо определяют путём статистической обработки «наблюдений» модели.

- **Применяется для интегралов большой размерности**
 - Например для одномерной функции достаточно разбиения на 10 отрезков и вычисление 10 значений функции (см. метод прямоугольников)
 - Если функция n -мерная (задачи теории струн и т.д.), то по каждой размерности разбиваем на 10 отрезков, следовательно потребуется 10^n вычислений значения функции

- **Суть метода МК (для одномерного случая)**

1. Бросаем n точек, равномерно распределённых на $[a, b]$, для каждой точки u_i вычисляем $f(u_i)$
2. Затем вычисляем выборочное среднее

$$\frac{1}{n} \sum_{i=1}^n f(u_i)$$

3. Получаем оценку интеграла

$$\int_a^b f(x) dx \approx \frac{b-a}{n} \sum_{i=1}^n f(u_i)$$

Точность оценки зависит только от количества точек n

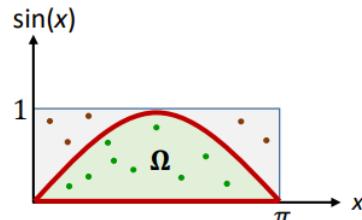
- Вычислить двойной интеграл методом Монте-Карло

$$I = \iint_{\Omega} 3y^2 \sin^2(x) dx dy, \quad \Omega = \{x \in [0, \pi], y \in [0, \sin(x)]\}$$

- Выберем n псевдо-случайных точек (x_i, y_i) , равномерно распределенных в области Ω
- Из общего числа n точек n' попали в область Ω , остальные $n - n'$ оказались вне области
- При значительном числе n интеграл приближенно равен

$$I \approx \frac{V}{n'} \sum_{i=1}^{n'} f(x_i, y_i)$$

$$f(x, y) = 3y^2 \sin^2(x), \quad V = \int_0^\pi \sin(x) dx = 2$$



два способа распараллеливания:

- По отрезкам, то есть каждый вычислитель генерирует n точек на определённом отрезке
- По количеству генерируемых точек, то есть каждый вычислитель генерирует $n / (\text{кол-во вычислителей})$ точек на всём отрезке интегрирования

Распараллеливание

Область интегрирования разбивается на подобласти по числу используемых процессов, а исходный интеграл представлялся в виде суммы интегралов по таким частичным отрезкам.

Схема распараллеливания

- Каждый из p процессов генерирует и бросает n / p точек
- Глобальная сумма формируется в корневом процессе

ИТОГО

Метод Монте-Карло обладает высокой степенью параллелизма при его реализации как на MPI, так и на OpenMP, программы на его основе позволяют равномерно загружать вычислительные узлы кластерных систем.

Недостатками являются:

- Границы погрешности не определены точно, но включают некую случайность. Это, однако, более психологическая, чем реальная, трудность.
- Статическая погрешность убывает медленно.
- Необходимость иметь случайные числа.

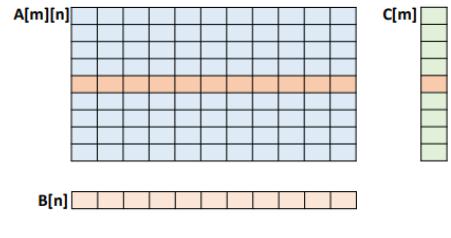
ПДФИНА КУРНОСЫЧА

13. Модель передачи сообщений. Подходы к распараллеливанию алгоритмов матричных вычислений: алгоритм умножения матрицы на вектор.

Данный алгоритм состоит в распараллеливании суммы частичных произведений элемента строки матрицы на элемент вектора $\sum a_{ij} \cdot b_j = c_i$

1. Размещение входных данных в памяти

- Массивы A, B и C размещены в памяти каждого процесса (помещаются в память одного процесса)
- Массивы хранятся в распределенном виде (не помещаются в память одного процесса)



2. Инициализация входных данных

- Данные инициализирует (загружает) один процесс и рассыпает всем
- Каждый процесс самостоятельно инициализирует (загружает) входные данные

B[n]

```
for (int i = 0; i < m; i++) {  
    c[i] = 0.0;  
    for (int j = 0; j < n; j++)  
        c[i] += a[i * n + j] * b[j];  
}
```

3. Параллельные вычисления

4. Формирование результата – вектора С

- Вектор C хранится в распределенном виде – у каждого процесса своя часть строк
- Собирается в корневом процессе
- Собирается во всех процессах (при условии достаточного объема памяти)

1. Массивы A, B и C размещены в памяти каждого процесса

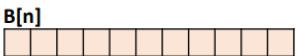
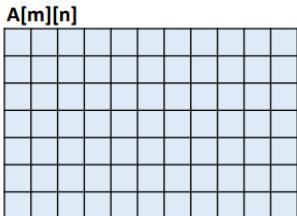
Определим предельные размеры матрицы и векторов

- Пусть матрица A квадратная ($m = n$)
- Элементы матрицы имеют тип double (8 байт)
- На вычислительном узле доступно d байт памяти
- Найдем наибольшее значение n , при котором массивы помещаются в доступную память

$$8(n^2 + 2n) = d$$

$$n = \sqrt{d/8 + 1} - 1$$

Массивы A, B и C размещены в памяти каждого процесса

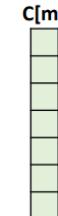


```

for (int i = 0; i < m; i++) {
    c[i] = 0.0;
    for (int j = 0; j < n; j++)
        c[i] += a[i * n + j] * b[j];
}

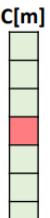
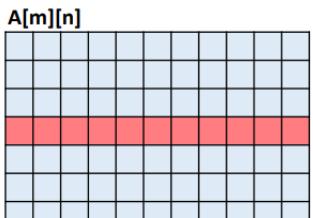
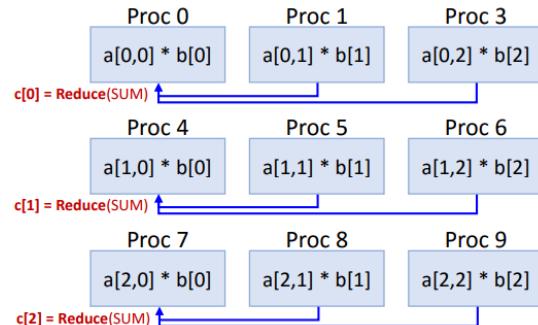
```

$$\begin{aligned}
c_0 &= a_{0,0}b_0 + a_{0,1}b_1 + \dots + a_{0,n-1}b_{n-1} \\
c_1 &= a_{1,0}b_0 + a_{1,1}b_1 + \dots + a_{1,n-1}b_{n-1} \\
&\dots \\
c_{m-1} &= a_{m-1,0}b_0 + a_{m-1,1}b_1 + \dots + a_{m-1,n-1}b_{n-1}
\end{aligned}$$



Вариант 1. Параллельное вычисление произведений $a_{ij}b_j$

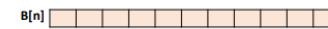
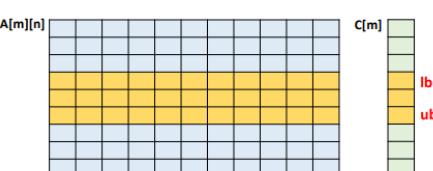
- Требуется $m * n$ процессов для вычисления произведений и n редукций для суммирования результатов в c_i
- Мелкозернистный параллелизм (fine grained)
- Результат хранится в分散ном виде: $c[0]$ в P0, $c[1]$ в P4, $c[3]$ в P7 (требуется еще один {All}Gather для сборки вектора)



```

for (int i = 0; i < m; i++) {
    c[i] = 0.0;
    for (int j = 0; j < n; j++)
        c[i] += a[i * n + j] * b[j];
}

```



```

for (int i = lb; i <= ub; i++) {
    c[i] = 0.0;
    for (int j = 0; j < n; j++)
        c[i] += a[i * n + j] * b[j];
}

```

Вариант 2. Каждый процесс вычисляет один элемент вектора

$$c_i = a_{i,0}b_0 + a_{i,1}b_1 + \dots + a_{i,n-1}b_{n-1}$$

- Плюсы: Не требуется дополнительных редукций
- Минусы: требуется m процессов
- Результат хранится в分散ном виде: $c[0]$ в P0, $c[1]$ в P1, $c[2]$ в P2 и т.д.
- Для сборки вектора требуется выполнить {All}Gather

$$\begin{aligned}
c_0 &= a_{0,0}b_0 + a_{0,1}b_1 + \dots + a_{0,n-1}b_{n-1} \\
c_1 &= a_{1,0}b_0 + a_{1,1}b_1 + \dots + a_{1,n-1}b_{n-1} \\
&\dots \\
c_{m-1} &= a_{m-1,0}b_0 + a_{m-1,1}b_1 + \dots + a_{m-1,n-1}b_{n-1}
\end{aligned}$$

Вариант 3. Каждый процесс вычисляет несколько значений вектора c_i

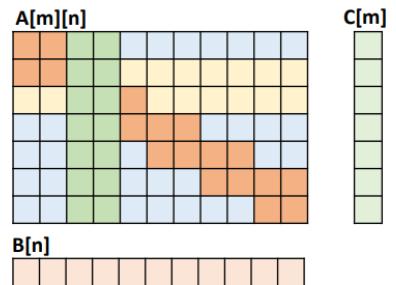
(m элементов вектора распределяются между p процессами)

- Каждый процесс вычисляет порядка m / p элементов вектора $c[m]$
- Крупнозернистый параллелизм (крупноблоочное распараллеливание, coarse-grained)
- Результат хранится в分散ном виде ($c[lb..ub]$)
- Для сборки вектора требуется выполнить {All}Gatherv

В первом варианте результирующий вектор $c[m]$ собирается в корне и надо делать $\text{rank} == 0$, а если использовать второй вариант, то достаточно использовать AllGatherv, а не просто Gatherv

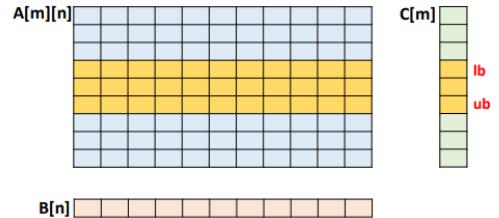
2. Матрица А хранится в распределенном виде

- Каждый процесс хранить часть матрицы А и векторов В и С (матрица A[m, n] может не помещаться в память узла)
- Варианты декомпозиции матрицы
 - Горизонтальными полосами – каждый процесс хранит часть строк матрицы А
 - Вертикальными полосами – каждый процесс хранит часть столбцов матрицы А
 - Диагональными полосами
- Варианты инициализация массивов
 - Корневым процессом + рассылка (суб)массивов всем процессам
 - Каждый процесс инициализирует массивы сам
- Варианты формирование результат – вектора C[m]
 - Хранится в распределенном виде (сборка не требуется, если памяти узла не достаточно для хранения всего вектора C[m])
 - Сборка в корневом процессе (Gather{v})
 - Сборка во всех процессах (Allgather{v})



2. Матрица А хранится в распределенном виде

- Каждый процесс хранит часть матрицы А и оба вектора В и С
- Матрица А разбивается на горизонтальные полосы
- Потребление памяти процессом: $O(mn / p + n + m)$
- Каждый процесс сам инициализирует векторы В и С и подматрицу А
- Результат собирается в корневом процессе



Найдем наибольшее значение n , при котором массивы помещаются в доступную распределенную память кластера из 10 узлов с объемом памяти 8 Гб (из них доступно 80%)

- Пусть матрица А квадратная ($m = n$), элементы матрицы имеют тип double (8 байт)

$$8(n^2/10 + 2n) = d$$

$$n \approx 92\,671$$

(под массивы требуется примерно 64 Гб)

Пиздануть про то, что памяти может нехватить, если все будут хранить дох данных.

[ССЫЛКА НА ПДФ КУРНОСЫЧА](#)

14. Модель передачи сообщений. Подходы к распараллеливанию прямых методов решения систем линейных алгебраических уравнений: метод Гаусса.

Допустим, дана СЛАУ и требуется найти неизвестные x

- **Метод Гаусса** (Gaussian elimination, row reduction) – метод последовательного исключения переменных
- **Шаги метода Гаусса:**
 1. **Прямой ход** (elimination) – СЛАУ приводится к треугольной форме путем элементарных преобразований (вычислительная сложность $O(n^3)$)
 2. **Обратный ход** (back substitution) – начиная с последних, находятся все неизвестные системы (вычислительная сложность $O(n^2)$)

Параллельный метод Гаусса

Версия 1

- Каждый процесс хранит в своей памяти одну строку матрицы – одно уравнение
- Требуется n процессов

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{15}x_5 = b_1 \quad \text{Процесс 0}$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{25}x_5 = b_2 \quad \text{Процесс 1}$$

$$a_{31}x_1 + a_{32}x_2 + \dots + a_{35}x_5 = b_3 \quad \text{Процесс 2}$$

$$a_{41}x_1 + a_{42}x_2 + \dots + a_{45}x_5 = b_4 \quad \text{Процесс 3}$$

$$a_{51}x_1 + a_{52}x_2 + \dots + a_{55}x_5 = b_5 \quad \text{Процесс 5}$$

Процессы неравномерно загружены вычислениями – после передачи строки они выбывают из вычислений

- Прямой ход**
- Процесс 0 передает свою строку 1 всем
 - Процессы 1.. $P-1$ исключают x_1 из своих уравнений
 - Процесс 1 передает свою строку 2 всем
 - Процессы 2.. $P-1$ исключают x_2 из своих уравнений
 - ...
 - Процесс $P-2$ передает свою строку $n-1$ всем
 - Процесс $P-1$ исключают x_{n-1} из своего уравнения

- Прямой ход**
- Процесс $P-1$ вычисляет x_n и передает всем
 - Процесс $P-2$ вычисляет x_{n-1} и передает всем
 - ...
 - Процесс 1 вычисляет x_2 и передает всем
 - Процесс 1 вычисляет x_1 и передает всем

Версия 2

- Каждый процесс хранит в своей памяти горизонтальную полосу из n / P смежных строк
- Требуется P процессов

Схема прямого и обратного ходов такая же, как в версии 1

Процессы неравномерно загружены вычислениями – после передачи строки они выбывают из вычислений

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{15}x_5 = b_1 \quad \text{Процесс 0}$$
$$a_{21}x_1 + a_{22}x_2 + \dots + a_{25}x_5 = b_2 \quad \text{Процесс 0}$$
$$a_{31}x_1 + a_{32}x_2 + \dots + a_{35}x_5 = b_3 \quad \text{Процесс 1}$$
$$a_{41}x_1 + a_{42}x_2 + \dots + a_{45}x_5 = b_4 \quad \text{Процесс 1}$$
$$a_{51}x_1 + a_{52}x_2 + \dots + a_{55}x_5 = b_5 \quad \text{Процесс 2}$$

Версия 3 – циклическое распределение строк матрицы

- Каждый процесс хранит в своей памяти порядка n / P строк, векторы $b[n]$ и $x[n]$
- Строки распределены по циклической схеме для выравнивания вычислительной загрузки процессов

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{15}x_5 = b_1 \quad \text{Процесс 0}$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{25}x_5 = b_2 \quad \text{Процесс 1}$$

$$a_{31}x_1 + a_{32}x_2 + \dots + a_{35}x_5 = b_3 \quad \text{Процесс 2}$$

$$a_{41}x_1 + a_{42}x_2 + \dots + a_{45}x_5 = b_4 \quad \text{Процесс 0}$$

$$a_{51}x_1 + a_{52}x_2 + \dots + a_{55}x_5 = b_5 \quad \text{Процесс 1}$$

Алгоритм:

Сначала рассылаются строки соответствующими процессами, которые хранят их, затем остальные процессы (которые имеют строки ниже пересланной по порядку в СЛАУ) вычитают эту строку из своих.

Прямой проход заканчивается только в случае, когда все строки разосланы и получилась треугольная форма СЛАУ.

Далее идёт обратный ход. Снизу начинается расчёт и результат передаётся вверх по СЛАУ. Пока не дойдёт до x_1 .

Прикол алгоритма версии 3 в том, что процесс хранит не только одну строку в памяти, что позволяет избежать тупняка при отсылки её другим процессам. Строки распределены циклически.

[ПОДРОБНЫЙ АЛГОРИТМ В ПДФине](#)

15. Модель передачи сообщений. Подходы к распараллеливанию сеточных методов: решение стационарного двумерного уравнения Лапласа.

Расчет стационарного распределения тепла

- С течением времени в теле устанавливается некоторое не зависящее от времени распределение температуры (тепловое состояние выходит на стационарный режим)
- Распределение температуры в таком случае описывается уравнением теплопроводности
- Стационарное двумерное уравнение Лапласа (Laplace equation)

$$\Delta U = \frac{d^2U}{dx^2} + \frac{d^2U}{dy^2} = 0 \quad (1)$$

- Функция $U(x, y)$ – неизвестный потенциал (температура)
- Задано** уравнение (1), значения функции $U(x, y)$ на границах расчетной 2D-области
- Требуется** найти значение функции $U(x, y)$ во внутренних точках расчетной 2D-области

Расчет стационарного распределения тепла

- Найти решение стационарного двумерного уравнения Лапласа (Laplace equation)

$$\frac{d^2U}{dx^2} + \frac{d^2U}{dy^2} = 0$$

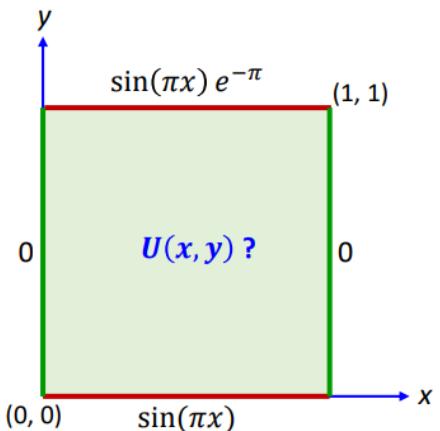
- Расчётная область (domain) – квадрат $[0, 1] \times [0, 1]$

- Границные условия (boundary conditions):

- $U(x, 0) = \sin(\pi x)$
- $U(x, 1) = \sin(\pi x) e^{-\pi}$
- $U(0, y) = U(1, y) = 0$

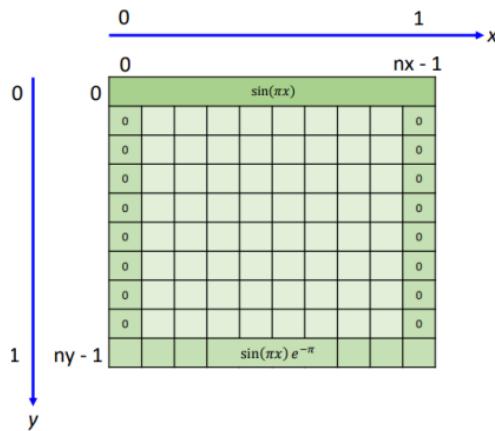
- Для данной задачи известно аналитическое решение

$$U(x, y) = \sin(\pi x) e^{-\pi y}$$



Дискретизация расчетной области

- Расчетная область $[0, 1] \times [0, 1]$ покрывается прямоугольной сеткой с постоянным шагом: n_x точек по оси ОХ и n_y точек по оси ОY



- Расчетная сетка – массив $[n_y, n_x]$ чисел (температура)
 - Переход от индекса ячейки $[i, j]$ к координатам в области $[0, 1] \times [0, 1]$:
- $$x = j * 1.0 / (n_x - 1.0)$$
- $$y = i * 1.0 / (n_y - 1.0)$$

Разностная аппроксимация оператора Лапласа

- Вторые производные аппроксимируются на расчетной сетке разностным уравнением с применением четырехточечного шаблона

$$\Delta U = \frac{d^2U}{dx^2} + \frac{d^2U}{dy^2} = 0$$

- Новое значение в каждой точке сетки равно среднему из предыдущих значений четырех ее соседних точек (схема «крест»)

```
grid_new[i, j] = (grid[i - 1, j] + grid[i, j + 1] +
                    grid[i + 1, j] + grid[i, j - 1]) / 4
```

sin(pi*x)							
0							0
0							0
0							0
0							0
0							0
0							0
0							0
sin(pi*x) e^-pi							

Короче можно эту всю канитель решить через метод Якоби (метод последовательных итераций) Прикол в распределении на области(квадраты).

Метод последовательных итераций Якоби (Jacobi)

- Вычисляем новое значение в каждой точке $[i, j]$ сетки – среднее из предыдущих значений четырех ее соседних точек (схема «крест»), результат записываем в новую сетку (массив)
- На следующей итерации текущей делаем новую сетку предыдущей итерации
- Заканчиваем итерационный процесс, если разность между каждым текущим и предыдущим значениями по модулю не больше EPSILON

Параллельная версия метода Якоби (1D)

- Одномерная (1D) декомпозиция расчётной области на горизонтальные полосы
- Каждому процессу назначается ny / p строк расчетной сетки
- Сетка хранится в памяти в распределенном виде
- Проблема – для расчета значений некоторых точек требуются данные соседних полос, которые находятся в памяти других процессов



Параллельная версия метода Якоби (1D)

- Анализ времени выполнения информационных обменов при одномерной декомпозиции
- Время передачи сообщения размером m байт: $t(m) = \alpha + \beta m$
- Пусть сетка квадратная и содержит n строк и столбцов
- На каждой итерации выполняется два send и два recv: $t_{iter} = 4\alpha + 4n\beta$

Параллельная версия метода Якоби (2D)

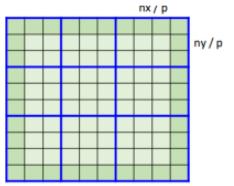
- Двумерная (2D) декомпозиция расчётной области на прямоугольные области
- Каждому процессу назначается подмассив $[ny / p, nx / p]$ строк расчетной сетки
- Сетка хранится в памяти в распределенном виде
- Проблема – для расчета значений некоторых точек требуются данные соседних полос, которые находятся в памяти других процессов



```

// Halo exchange: T = 4 * (a + b * (rows / py)) + 4 * (a + b * (cols / px))
thalo -= MPI_Wtime();
MPI_Irecv(&local_grid[IND(0, 1)], 1, row, top, 0, cartcomm, &reqs[0]);           // top
MPI_Irecv(&local_grid[IND(ny + 1, 1)], 1, row, bottom, 0, cartcomm, &reqs[1]);    // bottom
MPI_Irecv(&local_grid[IND(1, 0)], 1, col, left, 0, cartcomm, &reqs[2]);        // left
MPI_Irecv(&local_grid[IND(1, nx + 1)], 1, col, right, 0, cartcomm, &reqs[3]);   // right
MPI_Isend(&local_grid[IND(1, 1)], 1, row, top, 0, cartcomm, &reqs[4]);         // top
MPI_Isend(&local_grid[IND(ny, 1)], 1, row, bottom, 0, cartcomm, &reqs[5]);     // bottom
MPI_Isend(&local_grid[IND(1, 1)], 1, col, left, 0, cartcomm, &reqs[6]);       // left
MPI_Isend(&local_grid[IND(1, nx)], 1, col, right, 0, cartcomm, &reqs[7]);     // right
MPI_Waitall(8, reqs, MPI_STATUS_IGNORE);
thalo += MPI_Wtime();
// iterations

```



Это неблокирующие посылки и принятия посчитанных локальных сеточек от других процессов и потом ждём-с передачи всех в функции WaitAll

Параллельная версия метода Якоби (2D)

- Анализ времени выполнения информационных обменов при двумерной декомпозиции
- Время передачи сообщения размером m байт: $t(m) = \alpha + \beta m$
- Пусть сетка квадратная и содержит n строк и столбцов
- На каждой итерации выполняется четыре send и четыре recv:

$$t_{iter} = 8\alpha + \frac{8n}{p}\beta$$

[ПДФИНА КУРНОСЫЧА](#)

ЭВМ И ПЕРЕФИРИЙНЫЕ УСТРОЙСТВА

История вычислительной техники (механические и электромеханические ВМ)

- **Механические вычислительные машины.** Первые механические ЦВМ, предназначенные для выполнения арифметических операций, изобретены в 17 веке. Их появление в значительной степени явилось следствием общефилософской установки того времени, согласно которой в основе устройства мироздания лежат законы механики. Поэтому механические вычислительные машины должны были показать, что умственная деятельность человека также (хотя бы отчасти) может быть механизирована. Механические вычислительные машины были созданы В. Шиккардом (1623, Германия, не сохранились), Б. Паскалем (1642) и Г. В. Лейбницем (1672). В 18 веке Дж. Полени (1709, Италия), Ф. М. Ган (1774, Германия), Ч. Стенхуп (1775, Великобритания) и другие реализовали различные проекты вычислительных машин. Однако малая надёжность и высокая стоимость препятствовали их распространению.

В 1821 году в Париже Тома де Кольмар организовал первое мелкосерийное производство арифмометров, конструкция которых продолжала совершенствоваться почти до середины 20 века. К началу 20 века номенклатура выпускаемых вычислительных машин была уже достаточно велика, кроме арифмометров большим спросом пользовались и другие механические вычислительные машины, например простые и дешёвые карманные сумматоры Куммера (Россия, 1846), Ч. Г. Вебба (США, 1868). Подобные устройства

выпускались в разных странах вплоть до 1970 года. В 1884 году американская фирма NCR наладила производство кассовых аппаратов, которые надолго стали самым массовым видом вычислительных машин. Все эти машины применялись для решения достаточно простых задач с ограниченным объёмом вычислений.

Другой вид вычислительных машин - специализированные разностные машины, предназначались для табулирования функций, аппроксимированных полиномом n -й степени (где $n = 1, 2, 3 \dots$). Первым проектом такой вычислительной машины была разностная машина Ч. Бэббиджа (1821-33, не закончена). Созданные позднее разностные машины П. и Г. Шейцев (1853, Швеция) и М. Виберга (1863, Швеция) применялись для расчёта математических таблиц и были первыми вычислительными машинами, снабжёнными устройством для печати результатов. Они стали первыми вычислительными машинами, которые выполняли достаточно длинную последовательность арифметических операций автоматически. Известны также разностные машины Дж. Гранта (1876, США) и К. Гамана (1909, Германия).

Идея создания универсальной ЦВМ принадлежит Ч. Бэббиджу. В 1834 году он начал работу над проектом аналитической машины, первой вычислительной машины с программным управлением. Её конструкция, предвосхитившая структуру современных компьютеров, включала арифметическое устройство, устройство для хранения чисел, печатающее устройство. Вычисления должны были производиться специальным устройством в соответствии с программой, записанной на перфокартах. Леди Ада Лавлейс, написавшая несколько программ для аналитической машины, признана первым в мире программистом. Хотя проект Бэббиджа не был реализован, он послужил толчком к разработке других аналитических машин, в том числе механической - П. Ладгейта (1909, Великобритания, не построена) и электро-механической - Л. Торрес-и-Кеведо (Испания, 1914).

- **Электромеханические вычислительные машины.** К концу 19 века сложился достаточно широкий круг задач (экономических, статистических, научно-технических), требующих большого объёма вычислений. В 1889 году Г. Холлерит создал счётно-перфорационные машины (СПМ), первоначально предназначавшиеся для обработки статистической информации. Исходные данные (вручную с помощью перфоратора) переносились на перфокарты. Набор подготовленных перфокарт вводился в табулятор, который в автоматическом режиме считывал данные и выполнял необходимые вычислительные операции. Порядок выполнения операций задавался установкой электрических связей на коммутационной доске. Промежуточные результаты записывались в запоминающие регистры, окончательные результаты печатались или выводились на перфокарты. Счётно-перфорационные машины содержали арифметическое устройство, память (колода перфокарт и регистры для запоминания промежуточных результатов), устройства ввода (с перфокарт) и вывода данных, т. е. включали все элементы архитектуры автоматической ЦВМ. К 1930 году СПМ занимали доминирующее положение в области обработки больших массивов числовых данных, однако выполнение арифметических операций механическим способом ограничивало их производительность. В СПМ, как и в механической вычислительной машине, использовалась десятичная система счисления, исключением было только множительное устройство Т. Фаулера (1844, Великобритания), работавшее в уравновешенной троичной системе.

Первую попытку применить электромагнитное реле для построения ЦВМ предпринял А. Маркванд (США) в 1885 году, разработавший проект релейной логической вычислительной машины (не был реализован). В начале 1930-х годов, когда в системах телефонной связи уже широко применялись электромагнитные реле, было построено несколько специализированных релейных вычислительных машин. Вслед за ними - универсальные релейные вычислительные машины с программным управлением: двоичная машина Z-3 К. Цузе (1941), серия релейных машин Дж. Стибица (1940-46, США), десятичная машина Mark I Г. Айкена (1944). Их продолжали строить вплоть до конца 1950-х годов в ФРГ (К. Цузе), СССР (РВМ-1 Н. И. Бессонова, 1957) и других странах. Однако электромеханические вычислительные машины уже не могли обеспечить требуемую вычислительную мощность, и дальнейшее развитие вычислительных машин определила электронная техника.

Поколения ЭВМ.



Поколения ЭВМ

Поколения ЭВМ	I	II	III	IV
Годы применения	1946 - 1958	1958 - 1964	1964 – 1972	1972 – настоящее время
Основной элемент	Эл. лампы	транзистор	ИС	БИС
Количество штук в мире	десятки	тысячи	Десятки тысяч	Миллионы
Быстродействие (операций в секунду)	$10^3 - 10^{14}$	$10^4 - 10^6$	$10^5 - 10^7$	$10^6 - 10^8$
Носитель информации	Перфокарта, перфолента	Магнитная лента	Магнитный диск	Гибкий и лазерный диск
Размер ЭВМ	Большие	Значительно меньше	Мини-ЭВМ	Микро-ЭВМ

MyShared

Структура ЭВМ Фон Неймана

Архитектура фон Неймана — широко известный принцип совместного хранения команд и данных в памяти компьютера. Вычислительные системы такого рода часто обозначают термином «машина фон Неймана», однако соответствие этих понятий не всегда однозначно. В общем случае, когда говорят об архитектуре фон Неймана, подразумевают принцип хранения данных и инструкций в одной памяти.

Принципы фон Неймана

- **Принцип однородности памяти**

Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы. Распознать их можно только по способу использования; то есть одно и то же значение в ячейке памяти может использоваться и как данные, и как команда, и как адрес в зависимости лишь от способа обращения к нему. Это позволяет производить над командами те же операции, что и над числами, и, соответственно, открывает ряд возможностей. Так, циклически изменяя адресную часть команды, можно обеспечить обращение к последовательным элементам массива данных. Такой прием носит название модификации команд и с позиций современного программирования не приветствуется. Более полезным является другое следствие принципа однородности, когда команды одной программы могут быть получены как результат исполнения другой программы. Эта возможность лежит в основе трансляции — перевода текста программы с языка высокого уровня на язык конкретной вычислительной машины.

- **Принцип адресности**

Структурно основная память состоит из пронумерованных ячеек, причём процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые словами, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек — адреса.

- **Принцип программного управления**

Все вычисления, предусмотренные алгоритмом решения задачи, должны быть представлены в виде программы, состоящей из последовательности управляющих слов — команд. Каждая команда предписывает некоторую операцию из набора операций, реализуемых вычислительной машиной. Команды программы хранятся в последовательных ячейках памяти вычислительной машины и выполняются в естественной последовательности, то есть в порядке их положения в программе. При необходимости, с помощью специальных команд, эта последовательность может быть изменена. Решение об изменении порядка выполнения команд программы принимается либо на основании анализа результатов предшествующих вычислений, либо безусловно.

- **Принцип двоичного кодирования**

Согласно этому принципу, вся информация, как данные, так и команды, кодируются двоичными цифрами 0 и 1. Каждый тип информации представляется двоичной последовательностью и имеет свой формат. Последовательность битов в формате, имеющая определённый смысл, называется полем. В числовой информации обычно выделяют поле знака и поле значащих разрядов. В формате команды в простейшем случае можно выделить два поля: поле кода операции и поле адресов.

Количественные характеристики производительности

Быстродействие и производительность ЭВМ.

Вместе с тем, единица измерения быстродействия компьютера "операции в секунду" устарела. Она не достаточно правильно отражает быстродействие. Для компьютеров первых поколений под "операцией" часто понимали сложение двух целых чисел определенной длины. Операция умножения выполнялась в десятки раз медленнее, чем сложение. По этой причине для современных компьютеров чаще используется характеристика — тактовая частота. Тактовая частота - это количество импульсов в

секунду (герц), генерируемых тактовым генератором компьютера. Тактовая частота — более мелкая единица измерения, чем операции в секунду. Фирмы — производители компьютеров стремятся к тому, чтобы уменьшить количество тактов, необходимых для выполнения базовых операций, и, тем самым, повысить быстродействие компьютеров. Современные персональные компьютеры характеризуются быстродействием выше 2 Гц и ОЗУ — более 256 Мбайт. Определение характеристик быстродействия и производительности представляет собой очень сложную инженерную и научную задачу, до настоящего времени не имеющую единых подходов и методов решения.

Казалось бы, что более быстродействующая вычислительная техника должна обеспечивать и более высокие показатели производительности. При этом практика измерений значений этих характеристик для разнотипных ЭВМ может давать противоречивые результаты. Основные трудности в решении данной задачи заключены в проблеме выбора: что и как измерять. Укажем лишь наиболее распространенные подходы.

Одной из альтернативных единиц измерения быстродействия была и остается величина, измеряемая в MIPS (Million Instructions Per Second — миллион операций в секунду). В качестве операций здесь обычно рассматриваются наиболее короткие операции типа сложения. MIPS широко использовалась для оценки больших машин второго и третьего поколений, но для оценки современных ЭВМ применяется достаточно редко по следующим причинам:

- набор команд современных микропроцессоров может включать сотни команд, сильно отличающихся друг от друга длительностью выполнения;
- значение, выраженное в MIPS, меняется исходя из особенностей программ;
- значение MIPS и значение производительности могут противоречить друг другу, когда оцениваются разнотипные вычислители (к примеру, ЭВМ, содержащие сопроцессор для чисел с плавающей точкой и без такового).

При решении научно-технических задач в программах резко увеличивается удельный вес операций с плавающей точкой. Опять же для больших однопроцессорных машин в данном случае использовалась и продолжает использоваться характеристика быстродействия, выраженная в MFLOPS (Million Floating Point Operations Per Second — миллион операций с плавающей точкой в секунду). Для персональных ЭВМ данный показатель практически не применяется из-за особенностей решаемых задач и структурных характеристик ЭВМ.

5) Типы архитектур вычислительных систем

- Архитектура фон Неймана
- Многопроцессорная архитектура
- Многомашинная вычислительная система
- Архитектура с параллельными процессорами
- SISD
- SIMD
- MISD
- MIMD

6) Характеристики памяти ЭВМ

Характеристики:

- емкость (объем) - количество байтов памяти;
- быстродействие - время обращения к ячейкам памяти, определяемое временем считывания или времени записи информации. Измеряется в наносекундах (1010с);
- разрядность - количество линий ввода-вывода, которые имеют микросхемы оперативной и постоянной памяти или внешние накопители.

7) Форматы машинных команд

В команде, как правило, содержатся не сами операнды, а информация об объекте адресах ячеек памяти или регистрах, в которых они находятся. Код команды можно представить состоящим из нескольких полей, каждое из которых имеет свое функциональное назначение.

В общем случае команда состоит из:

- операционной части (содержит код операции);
- адресной части (содержит адресную информацию о местонахождении обрабатываемых данных и месте хранения результатов).

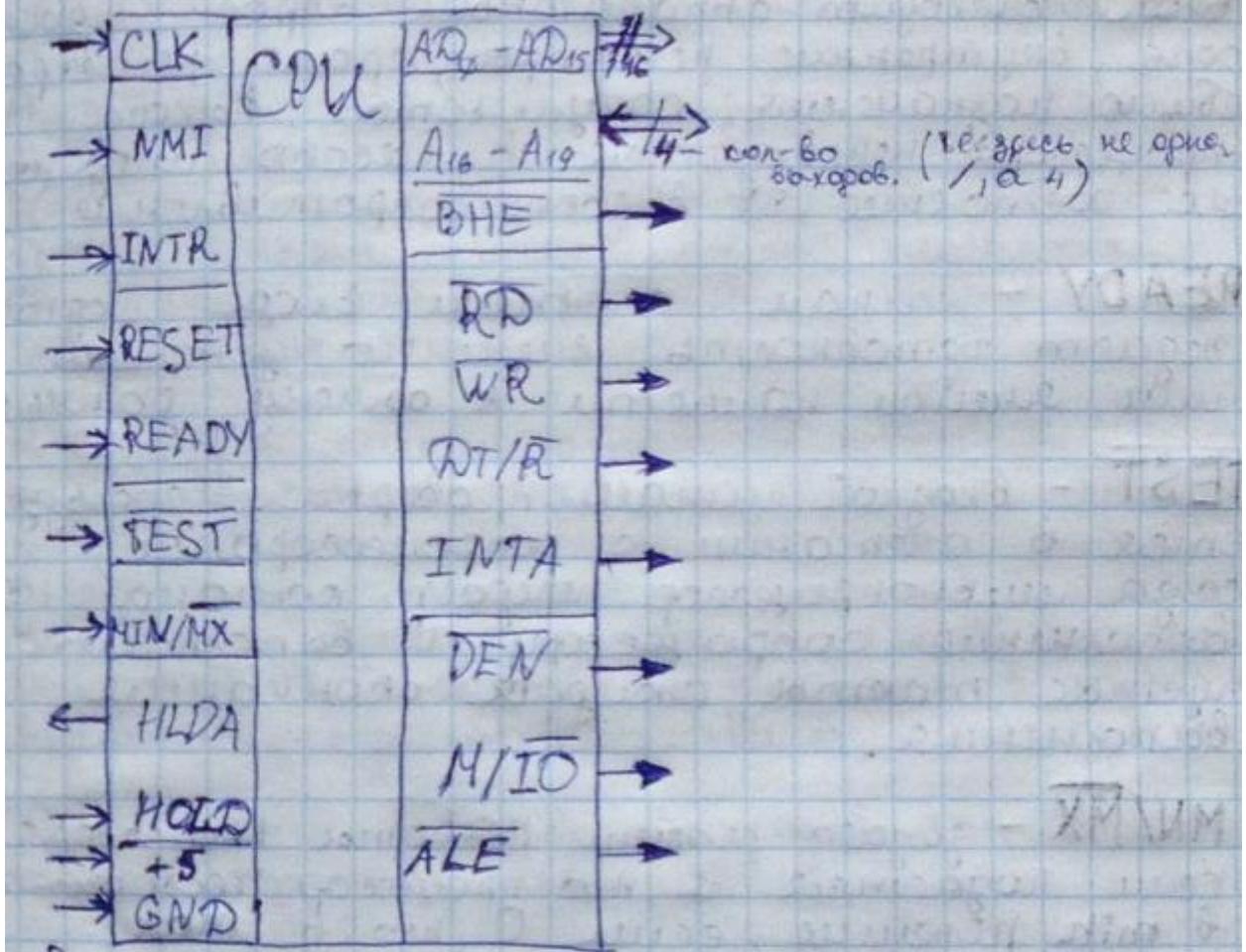
В свою очередь, эти части, что особенно характерно для адресной части, могут состоять из нескольких полей.

8) Способы адресации памяти

- Подразумеваемый operand
- Подразумеваемый адрес
- Непосредственная адресация
- Прямая адресация
- Относительная (базовая) адресация
- Укороченная адресация
- Регистровая адресация
- Косвенная адресация
- Адресация слов переменной длины
- Стековая адресация
- Автоинкрементная и автодекрементная адресации
- Индексная адресация

13. Назначение входов микропроцессора 8086

Назначение входов и выходов микропроцессора 40 pin.



Входы:

CLK – вход синхронизации, на который подаются тактовые импульсы, задающие частоту процессора.

NMI – вход немаскируемых запросов на прерывание. При поступлении запроса на этот вход он обрабатывается независимо от текущего состояния процессора.

INTR – вход маскируемых запросов на прерывание. При поступлении запроса на этот вход процессор проверяет состояние флага, разрешающего обработку прерывания, и если прерывание разрешены, то запрос обрабатывается, запрещены-игнорируется.

RESET – при поступлении сигнала на этот вход происходит аппаратный сброс процессора, внутренние регистры, кроме регистров общего назначения, обнуляются. Выходы процессора переводятся в высокоимпедансное состояние (т.е. высокое сопротивление).

READY – сигнал на этом входе подтверждает готовность внешнего устройства или ячейки памяти к обмену данными.

TEST – входной сигнал проверки. Используется в сочетании с сопроцессорами, когда микропроцессор передает команду на выполнение сопроцессору и выполняет пустые такты, ожидая окончания выполнения.

MN/MX – задает режим работы процессора. Если подается 1, то процессор работает в min режиме, если 0, то в max. Min режим используется в однопроцессорных конфигурациях,

также в сложных многопроцессорных. В **max** режиме большинство входов и выходов меняют свое назначение.

HOLD – вход запроса на прямой доступ к памяти. При поступлении сигнала процессор завершает выполнение текущей операции, передает управление контроллеру прямого доступа к памяти и отключается от внешних шин.

+5 – питание (+5В).

GND – заземление (отрицательный контакт питания).

14. Назначение выходов микропроцессора 8086

Схема входов и выходов, если нужна, находится в 13 вопросе

Выходы:

AD0-AD15 – совмешённая шина адреса и данных. В одни промежутки времени передаются адреса, в другие данные, т.е. используется временное разделение шины.

A16-A19 – старшие разряды шины адреса.

BHE – используется для разрешения подключения старшего банка памяти. 16-ти разрядная шина может использоваться и как 8-разрядная.

RD – 0-ой сигнал означает, что процессор осуществляет операцию чтения из памяти или внешнего устройства.

WR – выходной сигнал записи. Указывает на то, что МП выполняет цикл записи в память или внешнее устройство, и сопровождает данные, которые выдаются МП на шину данных. 0-й сигнал означает, что процессор осуществляет операцию записи в порт внешнего устройства или в память.

DT/R – выходной сигнал передачи/приема данных; определяет направление передачи данных. Предназначен для управления шинными формирователями и действует на протяжении всего цикла шины. 1 на выходе означает, что идет передача данных, 0 – означает, что идет прием.

INTA – используется для подтверждения обработки запроса на прерывание со входа INTR.

HLDA – выход подтверждения запроса на прямой доступ к памяти.

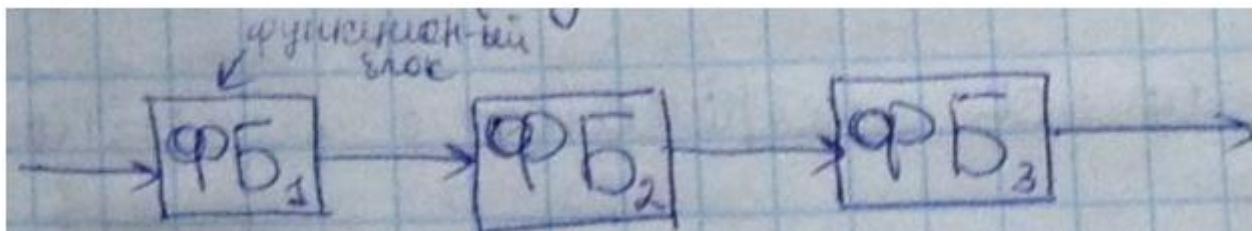
DEN – выходной сигнал разрешения данных, который определяет появление данных нашине адреса/данных. 0-ой сигнал свидетельствует о том, что по совмещеннной шине передаются данные.

ALE – выходной сигнал разрешения фиксации адреса; выдается в начале каждого цикла шины и используется для записи адреса в регистр-фиксатор

M/IO – выход (M- память, IO- порты ввода/вывода) разделения адресных пространств памяти и внешних устройств.

15. Конвейерные вычисления, общие понятия

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



Для согласования скоростей ФБ используют буферную память.



В простейшем случае в качестве простейшей памяти используется регистры.

Конвейеры делятся на 2 вида: синхронные и асинхронные.

В синхронных конвейерах начало выполнения любой операции определяется тактовыми импульсами.

В асинхронных обработка осуществляется по мере готовности исходных данных. Период тактовых импульсов определяется тах временем обработки самым медленным функциональным блоком и временем записи в буферную память.

Конвейеры бывают как линейные, так и нелинейные.

В линейном конвейере ФБ соединяются последовательно.

В нелинейном конвейере возможны дополнительные связи при этом допускаются обратные связи.

В большинстве случаев встречаются линейные конвейеры.

16. Конвейер команд

Является самым распространенным конвейером во всех современных процессорах.

Впервые конвейер команд был предложен в 1956 г. академиком Лебедевым.

Цикл команды представляет собой последовательность операций.

1 этап: выборка команды (ВК). Команда читается из памяти и заносится во внутренний регистр процессора.

2 этап: дешифрация команды (декодирование) (ДК). Определяется код операции и определяются способы адресации операнда.

3 этап: вычисление адресов operandов (ВА).

4 этап: выборка operandов (ВО). Команды читаются из памяти во внутренний регистр процессора.

5 этап: исполнение команды (ИК).

6 этап: запись результата (ЗР).

Лебедев предложил выделить все эти этапы в отдельные ступени конвейера.

Выполнение 9-и команд на конвейере:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
K ₁	VK	OK	VA	VO	IK	ZP									
K ₂	VK	OK	VA	VO	IK	ZP									
K ₃	VK	OK	VA	VO	IK	ZP									
K ₄		VK	OK	VA	VO	IK	ZP								
K ₅		VK	OK	VA	VO	IK	ZP								
K ₆		VK	OK	VA	VO	IK	ZP								
K ₇		VK	OK	VA	VO	IK	ZP								
K ₈		VK	OK	VA	VO	IK	ZP								
K ₉		VK	OK	VA	VO	IK	ZP								

Выполнение 9-ти команд на конвейере заняло 14 тактов. Без конвейера потребовалось бы 54 такта. В реальности такого существенного увеличения производительности достичь не удается.

Конфликтные ситуации в конвейере не позволяют достичь потенциальной производительности. Конфликты в конвейере также показывают рисками. Существует 3 причины конфликтов:

1) Структурный риск:

- несколько команд пытаются обратиться к одному и тому же ресурсу. (чаще всего этим ресурсом является память). Т.к. многие команды не предполагают обращения к памяти, влияние структурного риска на производительность не велико.

2) Риск по данным:

- две команды на разных ступенях конвейера работают с одной и той же переменной. Риск по данным является типичным для конвейера.

3) Риск по управлению:

- вызван неоднозначностью выполнения следующей команды после команды перехода (условного и безусловного). Это самая большая проблема для производительности конвейера.

Проблемы, связанные с безусловным переходом, решаются анализом команды. На этапе ее выборки, т.к. переход однозначный и адрес перехода известен.

Основную проблему составляет условный переход, т.к. до выполнения команды неизвестна необходимость перехода.

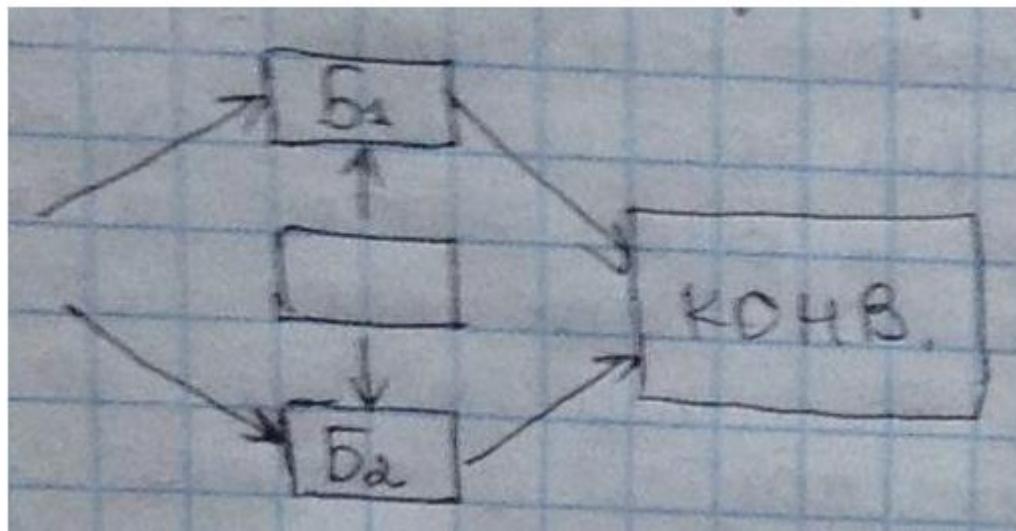
Методы решения проблем условного перехода

Для устранения и частичного сокращения издержек используется следующие подходы:

1. Использование буферов предвыборки.

На этапе декодирования команды распознаются. Наличие условного перехода.

Соответственно, становятся известны адреса, по которым следует переходить. В дальнейшем команды из памятичитываются в 2 буфера. В 1-ый буферчитываются команды, которые будут выполняться при отсутствии перехода. Во 2-ой буфер при его наличии.



Когда становится известен факт перехода содержимое одного из буферов отбрасывается, содержимое второго буфера выполняется.

2. Множественные потоки:

Часть ступеней конвейера дублируются. Аналогично 1-му подходу используются результаты одной из ветвей.

3. Задержанный переход:

Данный подход предполагает выполнение команд, следующих за командой условного перехода как будто такой команды и не было, естественно это имеет смысл только если следующие команды будут полезными. Основная нагрузка кладется на компиляторе, который старается вставить после команды условного перехода команды, которые будут выполнены в любом случае независимо от перехода.

4. Предсказание переходов:

Данный метод является наиболее эффективным способом борьбы с рисками по управлению. Идея такого подхода заключается в том, что еще до появления команды условного перехода либо сразу же при ее поступлении на конвейер делается предположение о наиболее вероятном исходе такой команды. Последующие команды подаются на конвейер согласно сделанному предположению. При ошибочном предсказании конвейер необходимо вернуть к состоянию до выборки ненужных команд. Цена ошибки предсказания достаточно высока (в плане производительности). Вводится понятие «точность предсказания». В современных подходах точность предсказания превышает 85%.

17. Конфликты в конвейере

Конфликтные ситуации в конвейере не позволяют достичь потенциальной производительности.

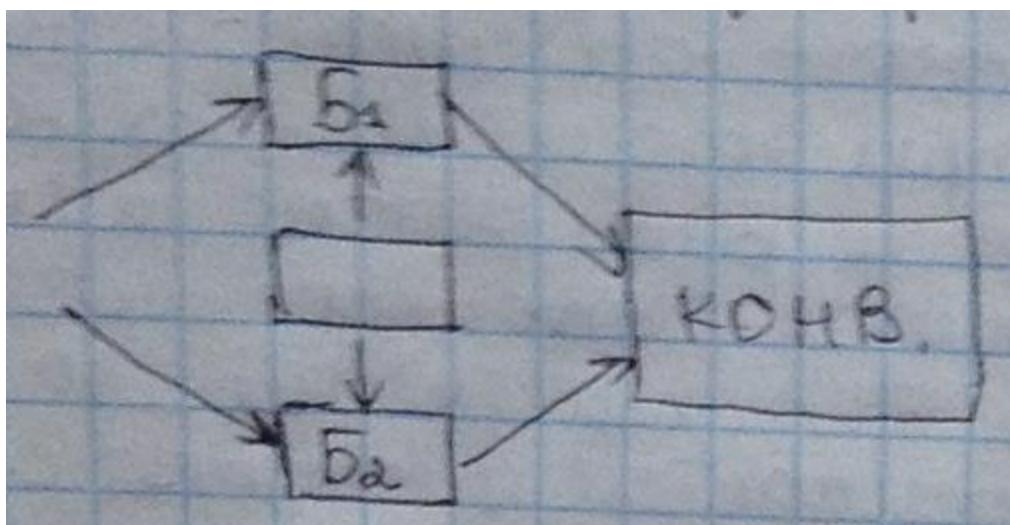
Существует 3 причины конфликтов:

- 1) **Структурный риск:** - несколько команд пытаются обратиться к одному и тому же ресурсу. (чаще всего этим ресурсом является память). Т.к. многие команды не предполагают обращения к памяти, влияние структурного риска на производительность не велико.
- 2) **Риск по данным:** - две команды на разных ступенях конвейера работают с одной и той же переменной. Риск по данным является типичным для конвейера.
- 3) **Риск по управлению:** - вызван неоднозначностью выполнения следующей команды после команды перехода (условного и безусловного). Это самая большая проблема для

производительности конвейера. Проблемы, связанные с безусловным переходом, решаются анализом команды. На этапе ее выборки, т.к. переход однозначный и адрес перехода известен. Основную проблему составляет условный переход, т.к. до выполнения команды неизвестна необходимость перехода.

Методы решения проблем условного перехода. Для устранения и частичного сокращения издержек используется следующие подходы:

1. Использование буферов предвыборки. На этапе декодирования команды распознаются. Наличие условного перехода. Соответственно, становятся известны адреса, по которым следует переходить. В дальнейшем команды из памятичитываются в 2 буфера. В 1-ый буферчитываются команды, которые будут выполняться при отсутствии перехода. Во 2-ой буфер при его наличии.



Когда становится известен факт перехода содержимое одного из буферов отбрасывается, содержимое второго буфера выполняется.

2. Множественные потоки:

Часть ступеней конвейера дублируются. Аналогично 1-му подходу используется результаты одной из ветвей.

3. Задержанный переход:

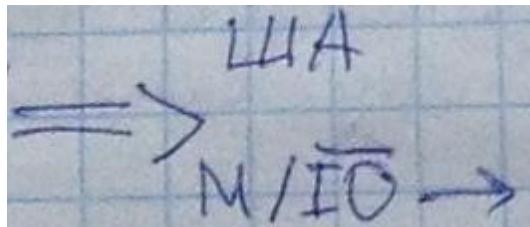
Данный подход предполагает выполнение команд, следующих за командой условного перехода как будто такой команды и не было, естественно это имеет смысл только если следующие команды будут полезными. Основная нагрузка кладется на компиляторе, который старается вставить после команды условного перехода команды, которые будут выполнены в любом случае независимо от перехода.

4. Предсказание переходов:

Данный метод является наиболее эффективным способом борьбы с рисками по управлению. Идея такого подхода заключается в том, что еще до появления команды условного перехода либо сразу же при ее поступлении на конвейер делается предположение о наиболее вероятном исходе такой команды. Последующие команды подаются на конвейер согласно сделанному предположению. При ошибочном

предсказании конвейер необходиимо вернуть к состоянию до выборки ненужных команд. Цена ошибки предсказания достаточно высока (в плане производительности). Вводится понятие «точность предсказания». В современных подходах точность предсказания превышает 85%.

18. Совмещение и разделение адресных пространств памяти и портов внешних устройств



Существует 3 варианта организации адресных пространств:

1.разделение:

В этом случае используется выход M\IO, который однозначно определяет принадлежность выставленного адреса. Адресные пространства полностью независимы.

Достоинства:

- 1) Все адреса доступны для памяти и для внешних устройств.
- 2) Во время выполнения команды однозначно известно осуществляется обращение к памяти или к внешнему устройству.

Недостатки:

- 1) Для работы с памятью существует большое количество команд, которые при данном способе организации адресных пространств недоступны для внешних устройств.
- 2) Существенное снижение количества способов адресации доступных для внешних устройств.

2.совмещение адресных пространств:

На выходе M/IO всегда присутствует 1, т.е. процессор считает, что он всегда работает с памятью, при этом часть адресного пространства отводится под порты ввода/вывода.

Достоинства:

- 1) Доступны все команды и способы адресации для портов внешних устройств.

Недостатки:

- 1) Часть адресного пространства съедается портами, соответственно эта память недоступна для адресации.

3.Гибридное использование адресного пространства:

Часть портов располагается в адресном пространстве памяти и к ним можно обращаться как к памяти, др. часть портов имеет выделенное адресное пространство, и не отнимает пространство памяти.

19. Иерархическая организация памяти ЭВМ

20. Шины ЭВМ

Все периферийные устройства подключаются к материнской плате через специальные разъёмы (рис. 13, 14), которые условно можно разделить на внешние и внутренние. Условность такого разделения объясняется тем, что некоторые внутренние разъёмы, используя специальные технические средства (кабели, планшеты и т.п.), могут стать внешними. Название разъёма совпадает с названием интерфейса (шины), через который будет передаваться информация между устройствами.

ISA (англ. Industry Standard Architecture – архитектура промышленного стандарта) – основная шина на компьютерах типа PC AT. Другое название этой шины – AT-Bus. Разрядность – 8 или 16 бит. Частота передачи данных – 8 Мбит/с. Максимальная пропускная способность – 16 МБ/с.

EISA (англ. Enhanced ISA – расширенная шина ISA) – функциональное и конструктивное расширение ISA. Внешние разъёмы имеют такой же вид, как и у ISA, и в них могут вставляться платы ISA. Но в глубине разъёма находятся дополнительные ряды контактов EISA, а платы EISA имеют более высокую ножевую часть разъёма с двумя рядами контактов, расположенных в шахматном порядке: одни чуть выше, другие чуть ниже. Разрядность – 32 бита, работает также на частоте 8 МГц. Предельная пропускная способность – 32 МБ/с.

VLB (англ. VESA Local Bus – локальная шина стандарта VESA) – 32-разрядное дополнение к шине ISA. Конструктивно представляет собой дополнительный разъём (116-контактный) при разъёме ISA. Разрядность – 32 бита, 16

тактовая частота – в диапазоне от 25 до 50 МГц. PCI (англ. Peripheral Component Interconnect – соединение внешних компонент) – является дальнейшим шагом в развитии VLB. Разрядность – 32 (расширенный вариант – 64) бита. Тактовая частота – до 33 МГц (PCI версии 2.1 – до 66 МГц). Пропускная способность шины – до 528 МБ/с (для 64-разрядной шины на 66 МГц).

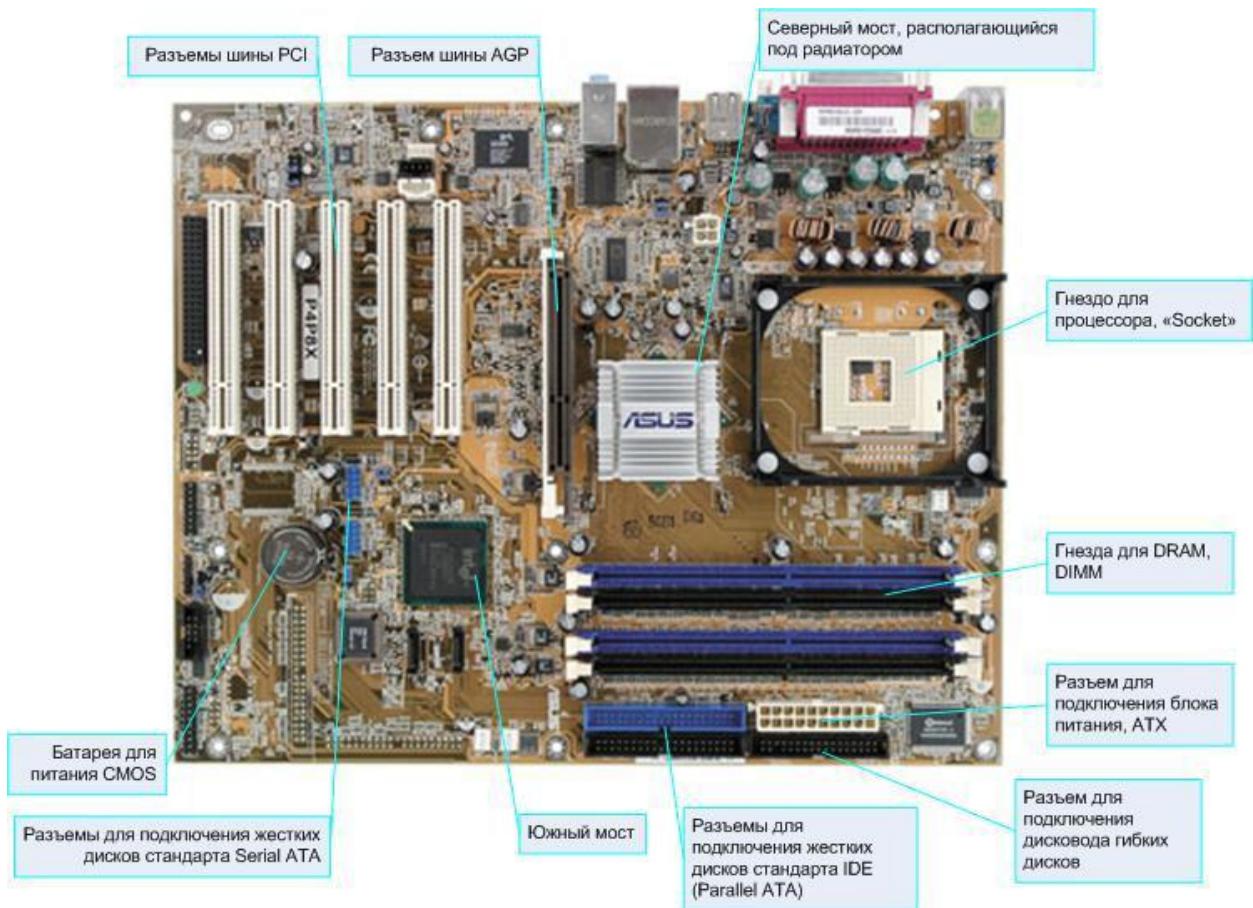


Рисунок 13. Элементы материнской платы на примере ASUS P4P8X



Рисунок 14. Внешние разъемы на примере материнской платы ASUS P4P8X

AGP (англ. Accelerated Graphics Port – ускоренный графический порт) является дальнейшим развитием PCI, нацеленным на ускоренный обмен графическими адаптерами с оперативной памятью. Пропускная способность увеличена за счёт разрядности, тактовой частоты и способа передачи данных по шине.

Последовательный порт (СОМ). Термин «последовательный» означает, что передача данных осуществляется побитно, используя один проводник. Такой тип связи характерен для телефонной сети, в которой для передачи данных в одном направлении используется один проводник. Управление последовательным портом осуществляется контроллером

UART (англ. Universal Asynchronous Receiver/Transmitter), который преобразует информацию из параллельного формата, используемого в ПК, в последовательный вид и наоборот. Термин «асинхронный» означает, что при передаче данных не используются никакие тактовые (синхронизирующие) сигналы, и передача данных может происходить через произвольные интервалы. Для того чтобы определить границы передаваемого блока информации, используются стартовый и стоповый сигналы (т.е. определённая последовательность нулей и единиц). Для подключения устройств используются 9- или 25-штырьковые разъёмы. Скорость обмена – до 115 Кбит/с.

Параллельный порт (LPT). Информация через такой интерфейс передаётся побайтно в параллельном режиме, т.е. для передачи данных в одну сторону применяются восемь проводников. Первоначально LPT был разработан для подключения печатающих устройств – принтеров. Подключение осуществляется с использованием 25-штырькового разъёма. Существуют одно- и двунаправленные параллельные интерфейсы.

PS/2 порты. Практически полный аналог СОМ-порта. Служат для подключения клавиатуры или манипулятора «мышь».

Универсальная последовательная шина (англ. Universal Serial Bus, USB). Является развитием последовательного интерфейса. Разрабатывалась для того, чтобы стало возможным подключать несколько устройств к одному порту и делать это без отключения ПК.

FDD (англ. Floppy Disk Driver – накопитель на гибких магнитных дисках). Конструктивно представляет собой 12x2-контактный игольчатый разъём с возможностью подключения двух дисководов через соединительный кабель – шлейф. В один момент времени информацию может передавать только один дисковод.

IDE (англ. Integrated Drive Electronics) или ATA (англ. AT Attachment). Используется для подключения устройств хранения информации (жёсткого диска, CD-ROM и т.п.). Конструктивно представляет собой 20x2-контактный игольчатый разъём, к которому через шлейф можно подключить до 2-х дисковых устройств. Чаще всего на материнской плате устанавливают 2 IDE контролера: Primary и Secondary. Существуют также

несколько протоколов обмена данными: UDMA/33 – 33 МБ/с и UDMA/66 – 66 МБ/с. Протокол UDMA/66 обладает вдвое большей скоростью передачи данных за счёт того, что данные передаются по обоим фронтам тактирующего сигнала в отличие от UDMA/33, вследствие чего необходим шлейф, в котором бы отсутствовали помехи от 2-х параллельно идущих проводников. Для решения этой проблемы применяется 80-жильный шлейф, каждый второй проводник которого соединён с общим проводом для уменьшения помех.

21.Что такое ЭВМ? Персональный компьютер? Корпус? Зачем используется блок питания? Зачем нужна материнская плата? Что такое форм-фактор? Что такое набор микросхем системной логики? Сколько шин в персональном компьютере? Зачем они нужны? Как определить пропускную способность шины? Виды памяти? Статическая и динамическая память? Что такое интерфейс? Какие интерфейсы используются в ПК?

21.1 Электронная вычислительная машина (ЭВМ) или компьютер (англ.

Computer

вычислитель) – это аппаратурно-программный комплекс, предназначенный для обработки информации. Под обработкой понимается: преобразование (т.е. выполнение некоторых вычислительных операций), а также ввод, вывод и хранение информации. Подобные устройства нашли применение практически во всех областях деятельности человечества. Изначально использовались большие сложные вычислительные машины, затем более широкое распространение получили персональные компьютеры.

21.2 Персональный компьютер (ПК) (англ. personal computer, PC) – это массово применяемая ЭВМ, имеющая небольшие габаритные размеры и невысокую стоимость. **Современный персональный компьютер** с архитектурой IBM PC (далее для краткости будем его просто называть персональный компьютер или ПК) в общем случае состоит из системного блока и внешних устройств (монитора, клавиатуры, манипулятора «мышь», колонок и т.п.). По сути, компьютером является то, что содержится в системном блоке, а именно:

- процессор;
- оперативная память;
- набор микросхем системной логики;
- слоты расширения.

Кроме этого, системный блок содержит корпус, блок питания, периферийные устройства (жёсткий диск, дисковод гибких дисков, CD-ROM и т.п.).

21.3 Корпус персонального компьютера

Корпус (рис. 7) – это кожух, внутри которого размещаются все основные компоненты персонального компьютера. Форм-факторы корпусов задают их геометрические размеры, форму и способы для крепления всего, что будет составлять системный блок, виды и расположение интерфейсных выводов, тип блока питания и способ его включения (тумблером или сигналом от материнской платы).

Корпусы бывают настольного (напольного) и стоечного исполнения. Первые предназначены для использования на рабочих местах, вторые для установки в специальные стойки, объединяющие несколько ПК и другое оборудование.

Корпусы с настольным исполнением могут располагаться горизонтально (типа «рабочий стол», англ. desktop) или вертикально (типа «башня», англ. tower).

21.4 Блок питания персонального компьютера

Назначение блока питания – преобразование электрической энергии, поступающей из сети переменного тока, в энергию, пригодную для питания узлов персонального компьютера. Входным может быть переменное напряжение с характеристиками 220 В / 50 Гц или 120 В / 60 Гц. Электронными схемами используется постоянное напряжение в + 3,3 В, ± 5 В и ± 12 В.

21.5 Системная (материнская) плата персонального компьютера

Системная плата (рис. 9) представляет собой многослойную плату, объединяющую все электрические схемы, которые, в совокупности, и образуют вычислительную машину. Кроме электрических схем на ней располагаются разъёмы для подключения процессора, памяти и периферийных устройств, а также некоторые переключатели. Структуру системной платы определяют состав смонтированных на ней электронных компонентов – **чипсет** (англ. *chipset*) и, конечно же, **форм-фактор**.

21.6 Form-factor

Все устройства, входящие в состав ПК, создаются на основе стандартов, называемых **форм-факторами** (англ. *form-factor*). Они определяют геометрические размеры изделия и интерфейс взаимодействия с другими узлами ПК.

На сегодняшний день существуют следующие форм-факторы материнских плат: AT (Baby-AT), ATX (miniATX, microATX, FlexATX), LPX, NLX (microNLX), WPX.

21.7 Набор микросхем системной логики (chipset)

Сама по себе материнская плата, с подключёнными к имеющимся на ней разъёмам внешними устройствами, ещё не является вычислительной машиной. Для организации взаимодействия между всеми этими узлами используется набор микросхем системной логики, называемый также **чипсетом** (англ. *chipset*). В его состав входят контроллеры прерываний, доступа к памяти, управления интерфейсами с внешними устройствами (последовательный и параллельный интерфейс, PCI, AGP, ISA, IDE и т.п.), часы реального времени и т.д. Состав и структура набора микросхем системной логики определяет основные характеристики ПК в целом.

В общем случае наборы микросхем системной логики строятся по принципу двух мостов – северного и южного. Использование двух мостов продиктовано соображениями увеличения производительности. К одному мосту подключаются устройства, медленно передающие информацию, к другому – передающие быстро. Между собой мосты взаимодействуют также через шину.

Северный мост (англ. *north bridge*) предназначен для организации взаимодействия между наиболее быстрыми узлами ПК, такими как процессор, оперативная память, ускоренный графический порт AGP (англ. *Accelerated Graphics Port*), интерфейс с внешними устройствами PCI (англ. *Peripheral Component Interconnect*, 33 МГц). Чаще всего он работает на полной тактовой частоте системной платы (на частоте шины процессора). В функции северного моста входит управление потоками данных из 4-х шин: шины памяти, AGP, системной шины процессора и шины связи с южным мостом. В современных наборах микросхем северный мост реализован в виде одной микросхемы (на одном кристалле), раньше для его реализации требовалось до трёх микросхем.

Южный мост (англ. *south bridge*) обслуживает медленные устройства, подключаемые через шину ISA (8 МГц), контроллер жёсткого диска IDE и USB (Universal Serial Bus). Также он содержит в себе схемы, реализующие функции энергонезависимой памяти системы ввода-вывода BIOS (англ. *Complementary Metal–Oxide–Semiconductor*, CMOS) и

часов реального времени, контроллер прямого доступа к памяти и контроллер прерываний.

В дополнение к мостам используется **третья микросхема, называемая Super I/O** и содержащая в себе контроллеры для управления некоторыми внешними устройствами (дисководом, последовательным и параллельным портами и т.п.).

21.8-10 Шины. Зачем. Пропускная способность. Сколько шин в персональном компьютере

Все устройства ПК, включая мосты, взаимодействуют между собой по принципу «общей шины», т.е. через один общий канал – шину, представляющую собой совокупность наборов информационных, адресных сигналов и сигналов управления (шина данных, шина адреса и шина управления). При этом в один момент времени передавать информацию по шине может только одно устройство, а все остальные подключённые к ней устройства могут только принимать или игнорировать данные.

Количество одновременно передаваемых сигналов называется **разрядностью** шины. Чем больше линий вшине (т.е. чем больше её разрядность), тем больше битов можно передать за одно и то же время. Таким образом, **скорость передачи данных** по шине (или её **пропускная способность**) определяется как произведение её тактовой частоты на разрядность.

- шина процессора;
- шина памяти;
- шина адреса;
- шины для подключения внешних устройств.

21.11 Виды памяти

В современных ПК для хранения информации используются следующие виды запоминающих устройств:

- динамическое запоминающее устройство с произвольным доступом (англ. dynamic random access memory, DRAM);
- статическое запоминающее устройство (англ. static random access memory, SRAM);
- постоянное запоминающее устройство (англ. read only memory, ROM);
- внешние устройства хранения данных (дисководы гибких дисков, жёсткие диски, CD-ROM, стримеры и т.п.).

Первые два типа памяти называются энергозависимыми, так как при отключении ПК информация в них теряется. Для долговременного хранения информации используются носители третьего и четвёртого типов.

Статическая и динамическая память используется для хранения оперативной информации. В *статической памяти* для организации ячеек используются специальные устройства – триггеры (построенные только на транзисторах), которые могут сохранять своё содержимое бесконечно долго, пока не будет выключено электропитание. За счёт отсутствия циклов регенерации и высокой скорости доступа к ячейкам статическая память значительно быстрее, чем динамическая. Однако триггеры намного дороже и больше по размерам, чем конденсаторы. Статическая память используется для организации буферов между быстродействующим процессором и медленной оперативной памятью.

Динамическая память оформляется в виде модулей, вставляемых в специальные разъёмы на материнской плате. Статическая память чаще всего выполняется в виде одной или

нескольких микросхем, располагаемых прямо на материнской плате. Кроме этого, современные процессоры содержат внутри статическую память малого размера.

22. Что такое таблица истинности? Булева функция? Как они связаны между собой? Как получить алгебраическую булеву функцию из таблицы истинности? И наоборот? Каким образом можно синтезировать логическую схему по таблице истинности? По алгебраической формуле?

22.1 Булева функция. Таблица истинности.

Чтобы математически описать схемы, которые строятся путём сочетания различных вентилей, используется особый тип алгебры, называемой булевой алгеброй, в которой все переменные и функции могут принимать только значения 0 или 1. Название эта алгебра получила в честь английского математика Джорджа Буля. Как и в обычной алгебре, правила преобразования входных значений в выходные называются функциями, которые могут зависеть от одной или нескольких переменных, и получать результат (0 или 1), основываясь только на их значениях. Если функция содержит n переменных, то существует 2^n возможных наборов значений функции. Булева функция может быть представлена двумя способами: в виде таблицы истинности и с помощью алгебраической записи.

Если написать таблицу, в которой перечислить всевозможные комбинации входных переменных и соответствующие этим комбинациям значения функций (выходное значение), то получится **таблица истинности**. Очевидно, что размер таблицы определяется числом переменных в функции и может быть огромным (2^n , где n – число входных переменных).

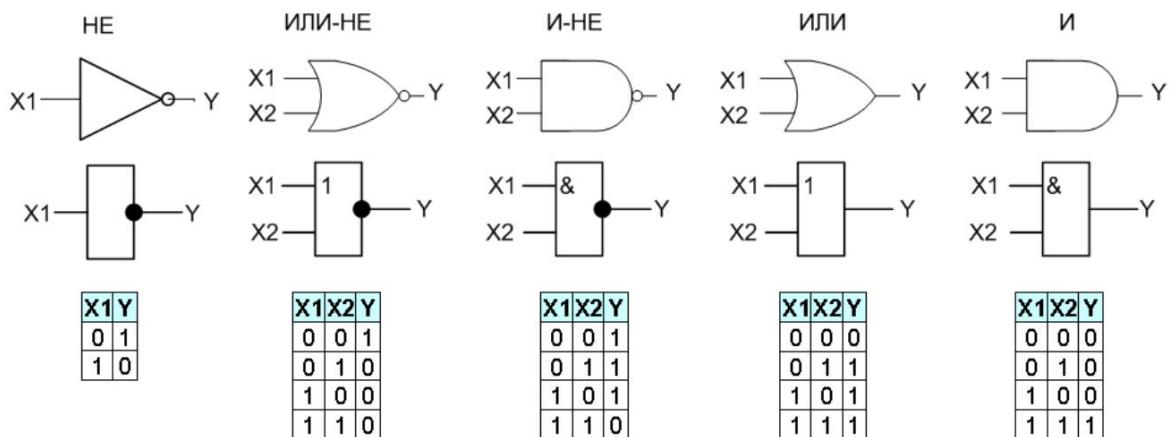


Рисунок 39. Значки для изображения вентилей на схемах и булевы функции, описывающие их работу (сверху вниз – европейское обозначение, российское обозначение, булева функция в виде таблицы истинности)

Как альтернатива таблицы истинности используется **алгебраическая запись**, в которой перечисляются все комбинации переменных, дающие единичное (или нулевое) значение функции. При этом знаком умножения обозначается операция И, а знаком сложения – операция ИЛИ, черта над переменной означает операцию НЕ. Например, для таблицы истинности вида:

X ₁	X ₂	X ₃	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

функция примет вид: $Y = \overline{X_1} \overline{X_2} \overline{X_3} + \overline{X_1} \overline{X_2} X_3 + X_1 \overline{X_2} \overline{X_3} + X_1 \overline{X_2} X_3$.

Здесь первое слагаемое образуется из инверсных значений входных переменных, так как единица на выходе соответствует их нулевым значениям, второе слагаемое – из двух инверсных и одного прямого, так как единица на выходе соответствует нулевым значениям двух переменных и единичному значению третьей и т.д.

22.2 Каким образом можно синтезировать логическую схему по таблице истинности

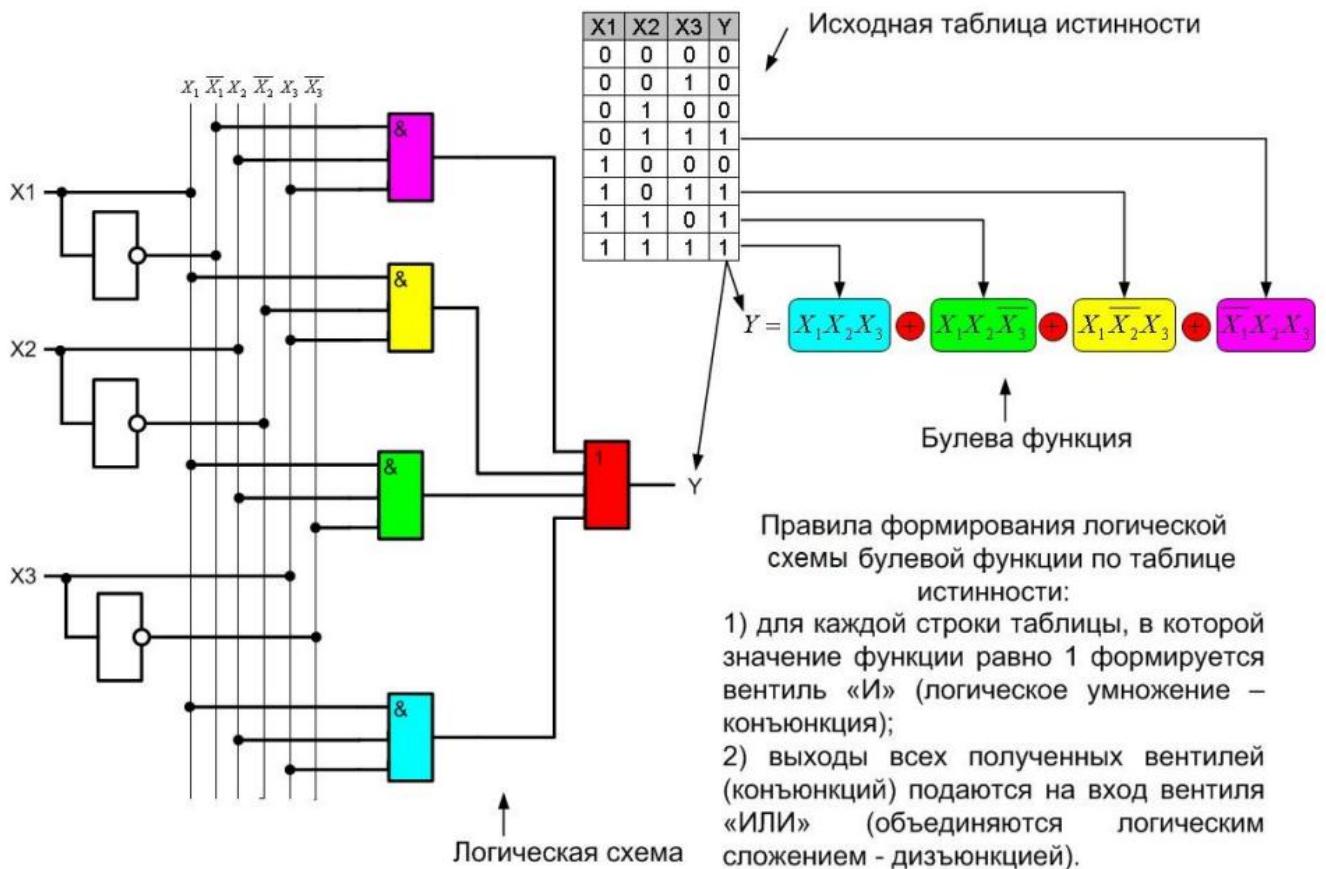


Рисунок 40. Формирование булевой функции и синтез логических схем

23. Что такое система счисления? Чем отличается позиционная система счисления от непозиционной? Как получить качественный эквивалент числа в непозиционной системе счисления? В позиционной? Как перевести числа из двоичной системы счисления в десятичную? Восьмеричную? Шестнадцатеричную? И наоборот?

Это вопрос по хранению информации ЭВМ. Любая информация в ЭВМ, включая текст, аудио и видео, представляется в виде последовательности нулей и единиц. Используя определённые правила, двоичные разряды формируются в числа, которые, в свою очередь, преобразуются в нужную для человека форму (текст, звук, изображение и т.п.).

23.1 Что такое система счисления

Для записи чисел человечеством придуманы различные правила, называемые системами счисления. По этим правилам любое число представляется в виде набора специальных символов – цифр. Получение количественного эквивалента числа осуществляется по алгоритму замещения, согласно которому сначала цифры заменяются их количественными эквивалентами, а затем эквивалент числа получается путём арифметических операций над эквивалентами цифр.

23.2 Чем отличается позиционная система счисления от непозиционной

В зависимости от того, меняет ли свое количественное значение цифра при разном положении в числе, системы счисления можно классифицировать как 1) *непозиционные* и 2)*позиционные* системы.

1) В системах счисления первого типа (непозиционных) число образуется из цифр, значение которых не изменяется при различном положении цифр в числе. Примером таких систем служит римская система счисления. В ней в качестве цифр для составления чисел используются буквы латинского алфавита. I означает единицу, V – пять, X – десять, L – пятьдесят, C – сто, D – пятьсот, M – тысячу. Для получения числа требуется просто просуммировать количественные эквиваленты входящих в него цифр, с учётом того, что если младшая цифра идет перед старшей цифрой, то она входит в сумму с отрицательным знаком. Например, DLXXVII = пятьсот + пятьдесят + десять + десять + пять + один + один = пятьсот семьдесят семь. Или CDXXIX = минус сто + пятьсот + десять + десять + минус один + десять = четыреста двадцать девять.

2) В позиционных системах счисления количественное значение цифры определяется её позицией в числе. Номер позиции называется *разрядом*. Число цифр, используемых для представления чисел, называется *основанием*. Количественное значение числа в позиционной системе счисления, состоящего из

$\{a_i\}$, $i \in \overline{0, n-1}$ (т.е. числа, имеющего вид $a_{n-1}a_{n-2} \dots a_1a_0$), может быть получено следующим образом:

$$A_{(p)} = a_{n-1}p^{n-1} + a_{n-2}p^{n-2} + \dots + a_1p^1 + a_0p^0,$$

число p называется *разрядностью* и определяет максимальный эквивалент, который можно получить для такого числа: $A_{(p)}^{\max} = p^n$.

В силу того, что ЭВМ строится на базе логических схем, которые могут иметь только два состояния – включено и выключено, то все числа в них представлены в двоичной системе счисления, которая по своей сути является позиционной. Набор цифр в этой системе

состоит из 0 и 1 (основание равно 2). Например, число в двоичной системе может иметь вид $10100111_{(2)}$. Количественный эквивалент такого числа равен ста шестидесяти семи ($167_{(10)}$).

23.3 перевод чисел из одной системы в другую

Перевод чисел из одной системы счисления в другую. Для выполнения арифметических операций над числами они должны быть представлены в одной системе счисления.

Перевод чисел из любой системы счисления в десятичную систему осуществляется простым получением их количественных эквивалентов и записи в виде десятичного числа. Перевод чисел из десятичной системы в любую другую осуществляется путём деления исходного числа на основание требуемой системы и записи остатков от деления в обратном порядке.

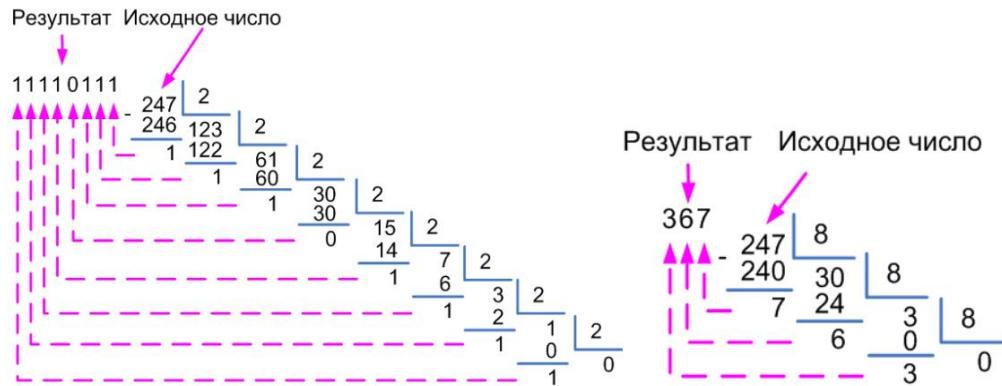


Рисунок 17. Перевод числа 247 из десятичной системы счисления в двоичную (слева) и восьмеричную (справа) системы

Например: Перевод числа из *шестнадцатеричной системы в двоичную* систему может быть осуществлён путём представления каждой его цифры в виде двоичной тетрады и последовательной записи этих тетрад. Например, число $FF_{(16)}$ представляется как $1111\ 1111_{(2)}$. А число $3A_{(16)}$ – как $0011\ 1010_{(2)}$.

Соответственно, перевод числа из *двоичной системы в шестнадцатеричную* систему осуществляется путём деления его на тетрады и представления каждой тетрады в виде шестнадцатеричной цифры. Например, $10111001_{(2)}$ представляется как $1011\ 1001_{(2)}$ и в шестнадцатеричной системе имеет вид $B9_{(16)}$.

Перевод числа из *восьмеричной системы в двоичную* систему осуществляется аналогично переводу числа из шестнадцатеричной системы, только цифры заменяются не тетрадами, а двоичными триадами. Например, $77_{(8)}$ будет представлено как $111\ 111_{(2)}$, а число $10_{(8)}$ – как $001\ 000_{(2)}$.

Двоичное число при переводе в восьмеричную систему счисления сначала делится на триады, а затем каждая триада представляется в виде восьмеричной цифры. Например, $11100111_{(2)}$, делится на $011\ 100\ 111_{(2)}$ и получается $347_{(8)}$.

24. Что такое флаг? Зачем он используется? Каким образом можно манипулировать флагами? Что такое маска? Как перевести числа из двоичной системы счисления в десятичную? Восьмеричную? Шестнадцатеричную? И наоборот? Что такое двоично-десятичное число? Какие базовые типы данных используются для хранения переменных в языке СИ?

24.1 Флаги. Маскирование

Это вопросы по теме **Разрядные и логические операции в языке Си.**

На практике часто приходится определять или задавать состояние устройств или управлять выполнением программы с использованием двоичных флагов – переменных, в которых может храниться только 0 или 1. Причем флаг считается установленным, если он содержит 1 и неустановленным – в противном случае. Примером использования флагов можно назвать программное управление выключателями: если необходимо выключатель перевести в состояние «включено», то устанавливаем флаг в 1, в противном случае – в 0. Конечно, для представления флага можно использовать одну целую переменную. Однако чаще всего приходится иметь дело одновременно с большим количеством флагов (управлять большим количеством выключателей). В этом случае целесообразно в качестве флага использовать один разряд целой переменной. Таким образом, при использовании 32-разрядных переменных, одной переменной может быть описано сразу 32 флага.

Именно такой способ применяется в современных процессорах для описания их состояния (регистр флагов – ячейка памяти, содержащая несколько двоичных разрядов-флагов). Для манипуляции отдельными битами целых переменных в языке Си используются поразрядные операции: И, ИЛИ, НЕ, ИСКЛЮЧАЮЩЕЕ ИЛИ, СДВИГ ВЛЕВО, СДВИГ ВПРАВО. Операция И обозначается символом &, операция ИЛИ – |, операция НЕ – ~, операция ИСКЛЮЧАЮЩЕЕ ИЛИ – ^, СДВИГ ВЛЕВО – <>. Выполнение этих операций происходит в двоичной системе счисления, несмотря на то, в каком виде представлены её операнды. Например, результат выполнения операции 2 & 5 будет равен 0 (т.к. 010 & 101 = 0). Чтобы получить значение определённого флага (т.е. располагающегося в определённом разряде числа) необходимо выполнить следующую последовательность действий:

flag = (registr >> (k - 1)) & 0x1,

где registr – это переменная, хранящая флаги, k – номер разряда (по порядку), в котором находится требуемый флаг. В результате этих действий переменная registr будет сдвинута вправо таким образом, что требуемый флаг окажется в младшем разряде, после чего будет произведена операция поразрядного И с единицей (т.е. все разряды числа, кроме младшего, будут умножены на 0). В переменную flag будет возвращена либо 1, либо 0, в зависимости от того, в каком состоянии был флаг.

Если требуется установить значение флага в единицу, то необходимо выполнить следующие действия:

registr = registr | (1 << (k - 1)).

Другими словами, необходимо выполнить операцию поразрядного ИЛИ между регистром флагов и числом, в котором в нужном разряде установлена 1, а в остальных разрядах числа содержатся нули.

Если требуется установить значение флага в нуль, то необходимо выполнить следующие действия:
 $\text{registr} = \text{registr} \& (\sim(1 << (k - 1)))$.

Другими словами, необходимо выполнить операцию поразрядного И между регистром флагов и числом, в котором в нужном разряде установлен 0, а в остальных содержатся единицы. Такое число получается путем инвертирования числа, содержащего единицу в том разряде, в котором нам нужен 0. Такая операция называется **маскированием**, а число, определяющее разряд (второй операнд) – **маской**. Ясно, что выделение разряда и установка его в нуль выполняется одинаковым образом, с тем лишь отличием, что в первом случае используется маска с единицей в требуемом разряде и остальными разрядами, равными нулю, а во втором случае – инверсная ей маска. Очевидно, что одновременно можно работать с несколькими разрядами. Необходимо только подобрать соответствующим образом второй операнд (маску).

24.2 Переводы из одной СС в Другую

Ответы на этот вопрос содержатся в 23 вопросе.

24.3 Двоично-десятичные числа

Двоично-десятичные числа – это специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры группой из четырех бит. При этом каждый байт содержит одну или две цифры. Первый способ называется неупакованное число, второй – упакованное. Например, число 3456 может быть записано как неупакованное в виде 00000011 00000100 00000101 00000110₍₂₎, или как упакованное – 0011 0100 0101 0110₍₂₎.

24.4 Какие базовые типы данных используются для хранения переменных в языке СИ?

В стандарте ANSI языка Си определены следующие базовые типы переменных:

Тип	Описание
int	Знаковый целый
char	Символьный
float	Вещественный одинарной точности с плавающей запятой
double	Вещественный двойной точности с плавающей запятой

Каждый из типов определяет формат хранения данных в переменных и, соответственно, диапазон допустимых значений, которые эти переменные могут принимать. Для изменения формата хранения данных используются модификаторы типа, определяющие наличие или отсутствие в переменной знака (signed или unsigned), увеличенный или сокращённый диапазон значений (long, long long (начиная со стандарта C99) или short). Для ввода и вывода значения переменных используются функции scanf и printf из стандартной библиотеки ввода-вывода. Чтобы определить, в каком виде выводить значения переменных, в функцию передаётся требуемый формат в виде последовательности символов % (процент) и буквы, определяющей необходимый формат: %d – означает вывести (или ввести) целое десятичное число со знаком; %o – целое без знака в восьмеричной системе счисления; %x – целое без знака в шестнадцатеричной системе счисления; %f – число с плавающей запятой и т.д. Например, если функция printf вызвана следующим образом: printf ("Значение x = %x\n", x);, то в стандартный поток вывода будет помещена фраза «Значение переменной x = », после чего туда же будет выведено значение переменной x в шестнадцатеричной системе счисления.

25. Взаимодействие с устройствами в Linux. Специальные файлы устройств. Функции open, close, read, write.

Каждое отдельное устройство, будь то дисковод, клавиатура или принтер, должно иметь свой **программный драйвер**, который выполняет роль транслятора или связующего звена между аппаратной частью устройства и программными приложениями, использующими это устройство.

В Linux драйверы устройств бывают трех типов:

- **Драйверы первого типа** являются частью программного кода ядра (встроены в ядро). Соответствующие устройства автоматически обнаруживаются системой и становятся доступны для приложений. Обычно таким образом обеспечивается поддержка тех устройств, которые необходимы для монтирования корневой файловой системы и запуска компьютера. Примерами таких устройств являются стандартный видеоконтроллер VGA, контроллеры IDE-дисков, материнская плата, последовательные и параллельные порты.
- **Драйверы второго типа** представлены модулями ядра. Они оформлены в виде отдельных файлов и для их подключения (на этапе загрузки или впоследствии) необходимо выполнить отдельную команду подключения модуля, после чего будет обеспечено управление соответствующим устройством. Если необходимость в использовании устройства отпадает, модуль можно выгрузить из памяти (отключить). Поэтому использование модулей обеспечивает большую гибкость, так как каждый такой драйвер может быть переконфигурирован без остановки системы. Модули часто используются для управления такими устройствами как SCSI-адаптеры, звуковые и сетевые карты.
Файлы модулей ядра располагаются в подкаталогах каталога `/lib/modules`. Обычно при инсталляции системы задается перечень модулей, которые будут автоматически подключаться на этапе загрузки. Список загружаемых модулей хранится в файле `/etc/modules`. А в файле `/etc/modules.conf` находится перечень опций для таких модулей. Редактировать этот файл "вручную" не рекомендуется, для этого существуют специальные скрипты (типа `update-modules`).
- И, наконец, для **третьего типа драйверов** программный код драйвера поделен между ядром и специальной утилитой, предназначеннной для управления данным устройством. Например, для драйвера принтера ядро отвечает за взаимодействие с параллельным портом, а формирование управляющих сигналов для принтера осуществляет демон печати `lpd` (который использует для этого специальную программу-фильтр). Другие примеры драйверов этого типа — драйверы модемов и X-сервер (драйвер видеоадаптера).

Но надо специально отметить, что во всех трех случаях непосредственное взаимодействие с устройством осуществляет ядро или какой-то модуль ядра. А пользовательские программы взаимодействуют с драйверами устройств через специальные файлы, расположенные в каталоге `/dev` и его подкаталогах. То есть

взаимодействие прикладных программ с аппаратной частью компьютера в ОС Linux осуществляется по следующей схеме:

устройство <-> ядро <-> специальный файл устройства <-> программа пользователя

Такая схема обеспечивает единый подход ко всем устройствам, которые с точки зрения приложений выглядят как обычные файлы.

Специальный файл устройства, или просто **файл устройства** (англ. special device file) — это один из типов файлов в UNIX-подобной операционной системе. **Специальные файлы устройств** содержат данные, необходимые операционной системе для взаимодействия с физическими устройствами, такими как диски и дисководы, принтеры и факсы и т. п. Фактически, **специальные файлы устройств** являются указателями на драйверы устройств, и когда процесс обращается к файлу устройств, он по сути работает с драйвером этого устройства.

Поскольку в операционной системе есть разные типы устройств, то и файлы устройств бывают разными. Есть два типа файлов устройств: **блочные** (англ. block special files) и **символьные** (англ. character special files). **Блочные файлы устройств** используются для передачи данных, разделённых на пакеты фиксированной длины — блоки. А **символьные файлы устройств** используются для небуферизованного обмена данными. Большинство устройств способно принимать и отправлять данные либо блоками (**блочные устройства**), либо сплошным потоком байтов (**символьные устройства**), но некоторые (такие как жёсткий диск) сочетают в себе обе эти возможности. Работа с первым типом устройств возможна либо через блочные, либо через символьные файлы, а вот с последним типом — подходят и те, и другие.

Для того, чтобы операционная система могла определить файл устройства и получить некоторые характеристики о самом устройстве, в файле содержатся 3 специальных поля: **класс устройства**, **старший номер устройства** и **младший номер устройства**. Класс устройства сообщает **символьное** устройство или **блочное**. В некоторых операционных системах (например, в Linux) есть и дополнительный класс устройств — небуферизованное символьное устройство. Кроме класса, есть ещё и тип устройства, который можно узнать по старшему номеру, например в Linux системе 1 означает оперативную память, 2 — дисковод гибких дисков, 3 — первый контроллер для жестких IDE-дисков, и т. д.. В разных операционных системах один и тот же старший номер может указывать на разные типы устройств. Для того, чтобы отличить два устройства одного класса и типа предусмотрели младший номер. Он используется для нумерации устройств с одинаковыми старшими номерами. Например, если в компьютере 2 одинаковых дисковода, то специальные файлы устройств для них будут содержать один и тот же класс устройства, один и тот же старший номер, но вот младший номер у одного из них будет 0, а у другого — 1.

Так как для пользователя драйвера выглядят как обычные файлы, к ним можно применять стандартные функции работы с файлами — **open**, **read**, **write**, **close**.

- **open(имя файла, флаги открытия(создание/чтение/запись), права доступа)** — открывает файл, возвращает при успешном открытии файловый дескриптор в виде некоторого числа, в противном случае возвращает 0.

- **read(файловый дескриптор, указатель на буфер для чтения, количество байт для чтения)** – считывает из файла.
- **write(файловый дескриптор, указатель на записываемый буфер, количество байт для записи)** – записывает в файл.
- **close(файловый дескриптор)** – закрывает открытый файл.

26. Терминалы. Типы терминалов. Эмуляция терминала. Режимы работы. Управление терминалом. Команды. Низкоуровневое управление.

Часто для взаимодействия с ЭВМ, в состав которых не предусмотрены собственные средства для взаимодействия с оператором (или они есть, но имеют скучный набор возможностей), используются специальные устройства, называемые **терминалами**.

Чаще всего **терминалы** образуют единое устройство, соединённое с ЭВМ (или каким-то другим устройством) через кабельные или телефонные каналы.

При этом к одной ЭВМ может подключаться несколько терминалов одно временно. Примерами терминалов могут служить: **POS-терминалы, операторские консоли** по управлению промышленным оборудованием и т.п.

Терминалы различаются возможностями устройств, входящих в их состав (т.е. сколько клавиш на клавиатуре, может ли монитор выводить графическую информацию или только текст, используется ли цвет для вывода информации на монитор и т.п.). Первые терминалы были вроде дистанционно управляемых пишущих машинок, которые могли только «отображать» (печатать на бумаге) символьный поток, посланный им из компьютера. Самые ранние модели назывались телетайпами, они могли выполнять перевод строки и возврат каретки точно так же, как обыкновенная пишущая машинка. Такие терминалы стали называть **пассивными, так как они только выводят информацию и не могут самостоятельно обрабатывать её**. Современные терминалы могут выполнять значительно больше действий, включающих ввод и вывод текстовой и графической информации, использование дополнительных каналов ввода информации (например, манипулятор «мышь»), выполнять дополнительные функции (например, распознавание штрих-кода) и т.п. Состав и структура терминала определяется областью применения. **Терминалы, способные проводить первичную обработку информации, называются активными**. Далее будет рассматриваться именно такие терминалы. Независимо от назначения и, соответственно, структуры терминала принцип его работы следующий:

Человек-оператор, нажимая клавиши на клавиатуре терминала, вводит информацию, которая поступает в устройство управления терминалом (УУТ). Далее она передаётся в ЭВМ в цифровом (численном) виде (скан-коды или управляющие последовательности), и поступает на вход специальной программе-драйверу, который её обрабатывает и передаёт выполняющейся программе (взаимодействующей с терминалом и ожидающей прихода от него данных). При необходимости информация, поступающая от клавиатуры, может сразу дублироваться на экране терминала. Такой режим называется «ЭХО».

Программа пользователя выводит результаты своей работы на терминал через драйвер, который передаёт её УУТ, а тот, в свою очередь, вырабатывает необходимые управляющие сигналы для монитора и выводит полученную информацию. Ввод информации через терминал может осуществляться в одном из двух режимов:

- **каноническом**, в котором информация передаётся в ЭВМ только в виде законченных строк (т.е. после нажатия клавиши «ВВОД» или «ENTER»);

- **неканоническом**, при котором вводимая информация сразу поступает в ЭВМ.

Помимо информационного потока между драйвером и УУТ передаются управляющие сообщения, которые позволяют настраивать УУТ на нужный режим работы и получать его текущие настройки. Программа пользователя также может управлять терминалом (например, передвинуть курсор на экране или изменить цвет выводимых символов) путём передачи ему специальных последовательностей символов, называемых командами терминала или управляющими кодами. Взаимодействовать напрямую с клавиатурой и монитором программы пользователя не может. Формат и назначение команд терминала определяются его типом и (обычно) описываются производителем терминала в руководстве пользователя и в соответствующей программной среде (например, в ОС семейства UNIX используется база данных настроек терминала, называемая *terminfo*). Управляющие коды (или управляющие символы) обычно состоят из первых 32 байтов алфавита ASCII. Они включают коды таких символов: возврат каретки (переместить курсор к левому краю экрана), перевод строки (переместить курсор вниз на одну строку), возврат на один символ, символ ESC, табуляция и звонок. Эти символы обычно не показываются на экране. Так как не имеется достаточного количества управляющих кодов, чтобы делать всё, используются команды терминала, чаще всего называемые Escape-последовательностями. Они состоят из нескольких подряд идущих символов, первым из которых является символ с кодом ASCII, равным 27, называемый «Escape» или ESC. Именно из-за первого символа команды терминала называются Escape-последовательностями. После получения символа ESC, УУТ исследует символы после него так, чтобы интерпретировать их и выполнить соответствующую команду. Когда распознаётся конец последовательности, дальнейшие полученные символы отображаются на экране (если дальше опять не следует команда). Некоторые Escape-последовательности могут иметь параметры (или аргументы), например, координаты на экране, куда надо переместить курсор. Параметры являются частью Escape-последовательности.

Современные персональные компьютеры имеют клавиатуру и монитор, непосредственно подключённые к их системному блоку. Функции по управлению этими устройствами возлагаются на операционную систему, которая воплощает в себе УУТ и драйвер терминала. Хотя производительность ПК достаточно высока, и возможности по вводу и выводу информации практически неограничены, они могут использоваться как терминалы для доступа к другим ПК или иным устройствам, к которым они подключены (например, видеосерверам, сетевым принтерам, модемам и т.п.). Для этого используются специальные программы, которые представляют ПК в виде псевдо- или виртуального терминала. Примером таких программ служат HyperTerminal, Remote Desktop, PuTTY, xterm и т.п. В некоторых операционных системах, например Linux, эмуляция терминала используется для того, чтобы позволить запустить несколько программ, выводящих различную информацию. При этом пользователь как будто работает за несколькими терминалами, поочерёдно переключаясь (например, нажимая комбинацию клавиш ALT+F1, или CTRL+ALT+F1) то на один, то на другой. Используя графический режим вывода информации на монитор и программы эмуляции текстовых терминалов, пользователь может одновременно запустить несколько виртуальных терминалов.

27. Как происходит обработка сигнала в программах, работающих под управлением ОС Linux?

При получении сигнала процесс может выполнить одно из трёх действий:

- выполнить действие по умолчанию. Обычно действие по умолчанию заключается в прекращении выполнения процесса. Для некоторых сигналов, например, для сигналов SIGUSR1 и SIGUSR2, действие по умолчанию заключается в игнорировании сигнала. Для других сигналов, например, для сигнала SIGSTOP, действие по умолчанию заключается в остановке процесса;
- игнорировать сигнал и продолжать выполнение. В больших программах неожиданно возникающие сигналы могут привести к проблемам. Например, нет смысла позволять программе останавливаться в результате случайного нажатия на клавишу прерывания, в то время как она производит обновление важной базы данных;
- выполнить определённое пользователем действие. Программист может задать собственный обработчик сигнала. Например, выполнить при выходе из программы операции по «наведению порядка» (такие как удаление рабочих файлов), что бы ни являлось причиной этого выхода.

Чтобы определить действие, которое необходимо выполнить при получении сигнала, используется системный вызов `signal`:

```
#include <signal.h>
typedef void (*sighandler_t) (int);
sighandler_t signal (int signum, sighandler_t handler);
```

Вызов `signal` определяет действие программы при поступлении сигнала с номером `signum`. Действие может быть задано как: адрес пользовательской функции (в таком случае в функцию в качестве параметра передаётся номер полученного сигнала) или макросы `SIG_IGN` (для игнорирования сигнала) и `SIG_DFL` (для использования обработчика по умолчанию). Если действие определено как пользовательская функция, то при поступлении сигнала программа будет прервана, и процессор начнёт выполнять указанную функцию. После её завершения выполнение программы, получившей сигнал, будет продолжено, и обработчик сигнала будет установлен в `SIG_DFL`. Чтобы вызвать сигнал, используется системный вызов `raise`:

```
#include <signal.h>
int raise (int signum);
```

Процессу посыпается сигнал, определённый параметром `signum`, и в случае успеха функция `raise` возвращает нулевое значение. Чтобы приостановить выполнение программы до тех пор, пока не придёт хотя бы один сигнал, используется вызов `pause`:

```
#include <unistd.h>
int pause (void);
```

Вызов `pause` приостанавливает выполнение вызывающего процесса до получения любого сигнала. Если сигнал вызывает нормальное завершение процесса или игнорируется процессом, то в результате вызова `pause` будет просто выполнено соответствующее действие (завершение работы или игнорирование сигнала). Если же сигнал перехватывается, то после завершения соответствующего обработчика прерывания

вызов pause вернёт значение – 1 и поместит в переменную errno значение EINTR. Пример программы, обрабатывающей прерывания:

```
1) #include <stdio.h>
2) #include <signal.h>
3)
4) /* Функция-обработчик сигнала */
5) void sghandler (int signo){
6)     printf ("Получен сигнал !!! Ура !!!\n");
7) }
8) /* Основная функция программы */
9) int main (void){
10)    int x = 0;
11)
12)    /* Регистрируем обработчик сигнала */
13)    signal (SIGUSR1, sghandler);
14)    do {
15)        printf ("Введите X = "); scanf ("%d", &x);
16)        if (x & 0x0A) raise (SIGUSR1);
17)    } while (x != 99);
18) }
```

28. Что такое прерывание? Что такое сигнал? Чем они отличаются друг от друга? Какую информацию несут в себе прерывание и сигнал?

Прерывание – это происходящее в ЭВМ, событие, при котором процессор временно приостанавливает выполнение одной (текущей) программы и переключается на выполнение другой программы, необходимой для обработки этого события. После окончания выполнения обработчика события, вызвавшего прерывание, процессор возобновляет выполнение приостановленной программы. Такой подход позволяет устройствам, входящим в состав ЭВМ, функционировать независимо от процессора и сообщать последнему о своей готовности взаимодействовать с ним. Кроме этого, использование прерываний позволяет реагировать на особые состояния, возникающие при работе самого процессора (т.е. при выполнении им программ).

Аналогом программных прерываний в UNIX-подобных операционных системах (например, Linux) служат сигналы. Сигнал – это способ взаимодействия программ, позволяющий сообщать о наступлении определённых событий, например, о появлении в очереди управляющих символов или возникновении ошибки во время работы программы (например, Segmentation Fault – выход за границы памяти). Как и аппаратное прерывание, сигналы описываются номерами, которые описаны в заголовочном файле signal.h. Кроме цифрового кода, каждый сигнал имеет соответствующее символьное обозначение, например, SIGINT. Большинство типов сигналов предназначены для использования ядром операционной системы, хотя есть несколько сигналов, которые посылаются от процесса к процессу.

Прерывания могут рассматриваться как средство связи между ЦП и ядром ОС. Сигналы можно рассматривать как средство связи между ядром ОС и процессами ОС.

Прерывания могут быть инициированы ЦП (исключения - например: деление на ноль, ошибка страницы), устройства (аппаратные прерывания - например, доступный вход) или инструкцией ЦП (ловушки - например, системные вызовы, точки останова). В конечном итоге им управляет процессор, который "прерывает" текущую задачу и вызывает обработчик ISR/прерывания с OS-ядром.

Сигналы могут инициироваться ядром ОС (например: SIGFPE, SIGSEGV, SIGIO) или процессом (kill()). В конечном итоге им управляет ядро ОС, которое доставляет их в целевой поток/процесс, вызывая либо общее действие (игнорирование, завершение, завершение и дамп-ядра), либо обработчик обработанного процесса.

Информацией, которую в себе несут, по большей части является сообщение о том, что какое-то событие произошло. В зависимости от события и вызывается определённое прерывание или сигнал.

29. Каким образом настраивается таймер? Как программа « узнаёт » о срабатывании таймера?

Работа с таймером. Как было сказано ранее, каждая программа в UNIX-подобных операционных системах может устанавливать три таймера:

ITIMER_REAL уменьшается постоянно и подаёт сигнал SIGALRM, когда значение таймера становится равным 0.

ITIMER_VIRTUAL уменьшается только во время работы процесса и подаёт сигнал SIGVTALRM, когда значение таймера становится равным 0.

ITIMER_PROF уменьшается во время работы процесса, и, когда система выполняет что-либо по заданию процесса. Совместно с ITIMER_VIRTUAL этот таймер обычно используется для профилирования времени работы приложения в пользовательской области и в области ядра. Когда значение таймера становится равным 0, подаётся сигнал SIGPROF.

Когда на одном из таймеров заканчивается время, процессу посыпается сигнал, и таймер обычно перезапускается. Для установки таймеров используется функция setitimer. Величина, которую устанавливается таймер, определяется следующими структурами.

Описание структур *itinterval* и *timeval*

```
struct itimerval {  
    struct timeval it_interval; /* следующее значение */  
    struct timeval it_value; /* текущее значение */  
};  
  
struct timeval {  
    long tv_sec; /* секунды */  
    long tv_usec; /* микросекунды */  
};
```

Значение таймера уменьшается от величины *it_value* до нуля, после чего генерируется соответствующий сигнал, и значение таймера вновь устанавливается равным *it_interval*. Таймер, установленный в нуль (его величина *it_value* равна нулю, или время вышло, и величина *it_interval* равна нулю), останавливается. Величины *tv_sec* и *tv_usec* являются основными при установке таймера. Если устанавливаемое время срабатывания таймера измеряется в полных секундах, то для установки таймера может использоваться системный вызов alarm:

Описание функций *alarm* и *setitimer*

```
#include <unistd.h>
unsigned int alarm(unsigned int secs);

#include <sys/time.h>

int setitimer(int which, const struct itimerval *value, struct
itimerval *oval);
```

Функция *alarm* запускает таймер, который через *secs* секунд сгенерирует сигнал *SIGALRM*. Поэтому вызов *alarm(60)*; приводит к посылке сигнала *SIGALRM* через 60 секунд. Обратите внимание, что вызов *alarm* не приостанавливает выполнение процесса, как вызов *sleep*, вместо этого сразу же происходит возврат из вызова *alarm*, и продолжается нормальное выполнение программы, по крайней мере, до тех пор, пока не будет получен сигнал *SIGALRM*. «Выключить» таймер можно при помощи вызова *alarm* с нулевым параметром: *alarm(0)*. Вызовы *alarm* не накапливаются. Другими словами, если вызвать *alarm* дважды, то второй вызов отменит предыдущий. Но при этом возвращаемое вызовом *alarm* значение будет равно времени, оставшемуся до срабатывания предыдущего таймера. Пример программы, настраивающей терминал и обрабатывающей сигнал от него, приведён в листинге ниже.

Листинг 5. Использование таймера

```
1) #include <stdio.h>
2) #include <signal.h>
3) #include <sys/time.h>
4)
5) void signalhandler (int signo){
6)     printf ("Сработал таймер\n");
7) }
8)
9) int main (void)
10) {
11)     struct itimerval nval, oval;
12)
13)     signal (SIGALRM, signalhandler);
14)
15)     nval.it_interval.tv_sec = 3;
16)     nval.it_interval.tv_usec = 500;
17)     nval.it_value.tv_sec = 1;
18)     nval.it_value.tv_usec = 0;
19)
20)     /* Запускаем таймер */
21)     setitimer (ITIMER_REAL, &nval, &oval);
22)
23)     while (1){
24)         pause();
25)     }
26)
27)     return (0);
28) }
```

30. Каким образом пользовательская программа может узнать об изменении размера окна виртуального терминала?

Управление терминалом производится либо управляющими воздействиями, либо установкой значений атрибутов терминала. Первый способ осуществляется с использованием вызова *ioctl* (Input Output ConTroL), второй – вызовами *tcsetattr* и *tcgetattr*.

Описание функции *ioctl*

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

int ioctl (int fd, int request, ...);
```

Вызов ioctl в качестве входных значений принимает номер дескриптора открытого файла терминала, которым надо управлять, номер требуемой операции и дополнительные параметры, необходимые для выполнения этой операции. Вызов ioctl является универсальным и не зависит от типа устройства (т.е. он применяется для управления не только терминалами). Поэтому сама функция ioctl описана в заголовочном файле sys/ioctl.h, а номера команд, которые она может выполнять над устройствами, – в других заголовочных файлах. Функции взаимодействия с терминалами описаны в заголовочном файле termios.h. Примером управляющего воздействия является определение размера экрана терминала. Результат работы ioctl помещается в структуру winsize, которая имеет вид:

Описание структуры winsize	
1)	struct winsize {
2)	unsigned short ws_row;
3)	unsigned short ws_col;
4)	unsigned short ws_xpixel;
5)	unsigned short ws_ypixel;
6)	}

Листинг 3. Определение размера экрана терминала	
1)	#include <stdio.h>
2)	#include <termios.h>
3)	#include <sys/ioctl.h>
4)	
5)	int main (void){
6)	struct winsize ws;
7)	
8)	if (!ioctl(1, TIOCGWINSZ, &ws)){
9)	printf ("Получен размер экрана.\n");
10)	printf ("Число строк - %d\nЧисло столбцов - %d\n",
11)	ws.ws_row, ws.ws_col);
12)	} else {
13)	fprintf (stderr, "Ошибка получения размера экрана.\n");
14)	}
15)	return (0);
16)	}

31. Основные этапы загрузки ПК на базе процессоров семейства Intel.

После нажатия кнопки «Power» источник питания выполняет самотестирование. Если все напряжения соответствуют номинальным величинам, то спустя некоторое время (примерно 0,1–0,5 с) он выдаёт на материнскую плату сигнал «PowerGood», после получения которого специальный триггер, вырабатывающий сигнал «RESET», снимает его с соответствующего входа микропроцессора. Далее сегментные регистры и указатель команд процессора устанавливаются в следующие состояния: CS = FFFFh; IP = 0; DS = SS = ES = 0. Все биты управляющих регистров и регистров арифметико-логического устройства устанавливаются в нулевое значение.

С момента снятия сигнала «RESET» микропроцессор начинает работу в реальном режиме, и, в течение примерно 7-ми циклов синхронизации, приступает к выполнению инструкции, считываемой из ROM BIOS и располагающейся по адресу FFFF:0000. В этой области памяти содержится только команда перехода на реально исполняемый код BIOS. В этот момент процессор не может выполнять никакую другую последовательность команд, поскольку нигде в любой из областей памяти, кроме BIOS, её просто не существует.

Последовательно выполняя команды BIOS, процессор реализует функцию начального самотестирования POST (англ. Power On Self-Test). На данном этапе

тестируются процессор, память и системные средства ввода-вывода, а также производится конфигурирование программно управляемых аппаратурных средств материнской платы. Обнаружив ошибку, система подаёт определённый звуковой сигнал.

В поисках встроенного драйвера видеоадаптера BIOS проверяет адреса памяти, начиная с 0000:0000 и заканчивая 0780:0000 (по умолчанию именно здесь должен располагаться такой драйвер). Если драйвер найден, проверяется контрольная сумма его кода, и, в случае совпадения с заданным значением, видеоадаптер инициализируется, и на экран выводится курсор. В противном случае на экране появляется сообщение вида «C00 ROM Error». Если встроенный драйвер видеоадаптера не найден, то используется видеодрайвер, записанный в ПЗУ материнской платы. Этот драйвер пытается стандартным образом инициализировать видеоадаптер и вывести на экран курсор. Если и это не срабатывает, то видеоадаптер считается неисправным, и подаётся соответствующий звуковой сигнал.

Далее сканируется память по адресам с C800:0000 по DF80:0000 с шагом 2 КБ в поисках встроенных драйверов любых других подключённых к материнской плате устройств (например, сетевых карт, модемов и т.п.). Обнаруженные драйверы выполняются так же, как и драйвер видеоадаптера. При несоответствии контрольных сумм выводится сообщение «XXXX ROM Eggog», где XXXX – сегментный адрес некорректного драйвера.

После инициализации всех устройств BIOS проверяет значение слова по адресу 0000:0472, в котором содержится информация, корректирующая процесс дальнейшей проверки системы. Если здесь записано значение 1234h, то дальнейшая проверка устройств (включая оперативную память) не производится. Это возможно только в случае «горячей» перезагрузки ПК (например, при нажатии комбинации клавиш CTRL+ALT+DEL). В обычном режиме или при «холодной» перезагрузке (т.е. при нажатии клавиши «Reset») здесь содержится значение 0000h.

В случае «горячей» загрузки BIOS проверяет остальные подключённые устройства. Часть конфигурирования выполняется однозначно, а другая часть может определяться положением перемычек (переключателей) системной платы или содержимым энергонезависимой памяти CMOS. Для изменения CMOS используется встроенная в BIOS утилита, называемая «Setup». Утилита «Setup» имеет интерфейс в виде меню или отдельных окон, ино-гда даже с поддержкой графики и мыши. Для запуска «Setup» во время выполнения POST появляется предложение нажать определённую комбинацию клавиш, например DEL.

После завершения POST BIOS определяет порядок поиска (англ. boot sequence) внешних устройств, чтобы загрузить операционную систему (ОС, специальную программу, управляющую работой ПК). Этот порядок определяется одним из параметров, содержащихся в CMOS. Например, если последовательность определена как A, C, D, то сначала будет проверен дисковод и, если в нём находится дискета, BIOS попытается использовать её для загрузки ОС. Если дискета не обнаружена, то будет проверен диск C, затем D.

После того, как определено устройство, с которого будет происходить загрузка операционной системы, BIOS считывает с него информацию, располагаемую в самом начале, в оперативную память по адресу 0000:7C00. После чего проверяется, является ли эта информация программой дальнейшей загрузки ОС. Если значения первых байтов считанного блока данных не корректны, на экране отображается сообщение об ошибке

загрузочной записи, и производится проверка следующего диска в списке. Если ни на одном из указанных носителей нет загрузочной программы, то на экран выводится сообщение об ошибке, и загрузка системы останавливается.

В случае корректности считанного блока данных начинается загрузка операционной системы, процедура которой зависит от типа ОС.

32. Геометрия жесткого диска. Что это такое? Трансляция геометрии. Типы трансляции.

Геометрией диска называется совокупность характеристик, позволяющих определить максимальный объём хранимой информацией. Другими словами, геометрия – это максимальное число цилиндров (C), число головок (H) и число секторов на дорожку (S).

$$\text{Ёмкость} = H \times C \times S \times (\text{размер сектора}).$$

При использовании линейной адресации секторов геометрией диска является всего лишь максимально возможный номер сектора. Для того чтобы сохранить принцип преемственности программного обеспечения (т.е. чтобы на новой аппаратуре могли работать старые программы), используется трансляция (рис. 34) CHS-адресации и геометрии в LBA и наоборот.

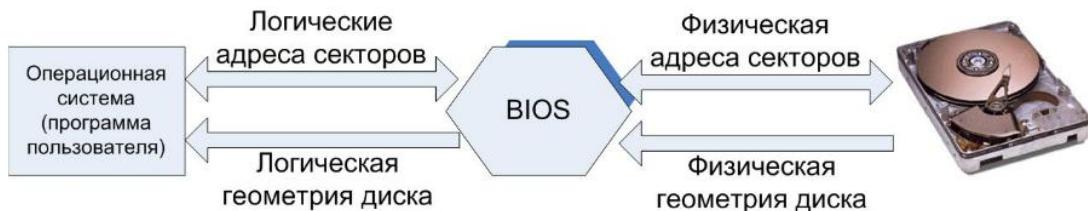


Рисунок 34. Принцип трансляции адресов секторов и геометрий жёстких дисков

CHS

Для того чтобы контроллер жёсткого диска считал или записал какой-то сектор, необходимо ему передать номер соответствующих головки, цилиндра и сектора. Такая адресация называется CHS (от англ. Cylinder/Head/Sector).

При этом способе сектор адресуется по его физическому положению на диске тремя координатами — номером цилиндра, номером головки и номером сектора.

LBA

При этом способе адрес блоков данных на носителе задаётся с помощью логического линейного адреса.

33. LBA адресация. Зачем используется. Перевод из LBA в CHSлог и наоборот.

LBA (logical block addressing) - логическая адресация блоков. Часто используется в качестве информации о количестве доступных физических секторов. Режим LBA использует линейную адресацию секторов, начиная с сектора 1, головки 0, цилиндра 0 как LBA 0 и заканчивая последним физическим сектором диска, который, например, на диске объемом 540 Мб имеет конечный номер LBA равный 1065456. Одним из преимуществ использования режима LBA - это уменьшение загрузки центрального процессора (CPU) поскольку операционная система адресует сектора линейно (LBA), и эти адреса обычно

пересчитываются в CHS (цилиндр-головка-сектор) для обращения к диску. При использовании же LBA, пересчета адресов не требуется.

Преобразование геометрии LBA к CHS:

1. Максимальный номер блока геометрии LBA делится на 63. $L = LBA / 63$.
2. Полученное число (L) делится на 1023, и округляется до одного числа из ряда: 2, 4, 8, 16, 32, 64, 128, 255. В результате получаем количество головок. $H = [L / 1023]$
3. Число L делится на число головок и получается число цилиндров. $C = [L / H]$

Пример:

$$LBA = 10018890.$$

1. $L = LBA / 63 = 159030$
2. $H_L = L / 1023 = \text{округляем}(155) = 255$
3. $C_L = L / H = 623$

$$CHS = 623/255/63$$

Трансляция адреса из LBA в CHS:

$$C = [[LBA / S_L] / H_L]; \quad H = [LBA / S_L] \% H_L; \quad S = LBA \% S_L + 1;$$

Трансляция адреса из CHS в LBA:

$$LBA = (C * H_L + H) * S_L + S - 1;$$

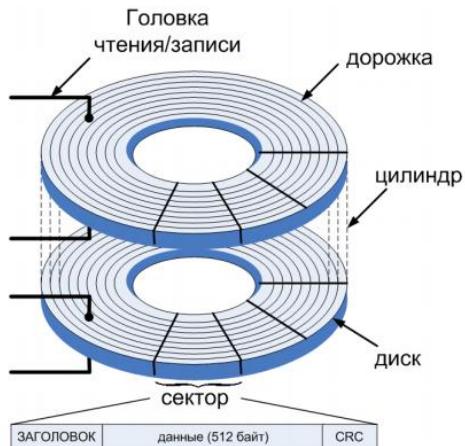
Где LBA – логический адрес в формате LBA, S_L – число секторов в логической геометрии, H_L – число головок в логической геометрии.

34. Адресация секторов жесткого диска. Типы адресации. Барьеры размеров дисков.

Почему возникли? Какие присутствуют?

Вся информация на диске записывается в форме концентрических окружностей, называемых дорожками. Расстояние между дорожками определяется шагом перемещения головок чтения-записи, которые располагаются над каждой поверхностью диска. Совокупность дорожек, расположенных друг над другом, называется цилиндром. Каждая дорожка поделена на дуги – секторы, которые являются минимальной единицей информации на жёстком диске.

Сектор состоит из трёх основных частей: заголовка, тела (512 байт), заключения. Информация в заголовке определяет начало и номер сектора. В заключении хранится контрольная сумма, необходимая для проверки целостности данных.



Для того чтобы контроллер жёсткого диска считал или записал какой-то сектор, необходимо ему передать номер соответствующей головки, цилиндра и сектора. Такая

адресация называется CHS (от англ. Cylinder/Head/Sector). Принято секторы нумеровать целыми числами, начиная с единицы, а цилиндры и головки – целыми числами, начиная с нуля.

Второй метод адресации, пришедший на смену CHS, LBA. LBA является более простым способом нумерации секторов, который не зависит от внутреннего устройства накопителя, где сектора адресуются только одним значением.

Стремясь сократить объём памяти, требуемый для адресации секторов на диске, в старых версиях BIOS были приняты ограничения для адресации секторов на диске. В них для «С» отводилось 10 бит, для «Н» – 8 бит, а для «S» – 6 бит. Для хранения CHS-адреса используется три 8-разрядных ячейки: первая – Н, вторая – 6 младших разрядов соответствуют номеру сектора, 2 старших – 2-м старшим разрядам номера цилиндра, третья ячейка – 8-ми младшими разрядами номера цилиндра.

Таким образом, получилось ограничение BIOS (или DOS, которая использовала только драйвер BIOS) в 8,5 ГБ. При использовании таких ограничений на величины С, Н, S был получен очередной «барьер размера жёсткого диска». Ограничения ATA и BIOS складываются так, что никто не может использовать больше чем 1024 цилиндра, 16 считающих головок и 63 сектора, что составляет 528482304 байт (504 МБ).

При адресации LBA выделялось гораздо больше памяти. В общем, максимальный объём жёсткого диска, с которым может работать LBA, составляет 128 Пиб.

35. Логическая организация винчестера. Разделы диска. Таблица разделов. Зачем используется. Структура.

Разметка — это разделение общего пространства диска на логические разделы иначе партиции, которые могут быть видны в операционной системе. Зачем вообще нужно такое разделение? Во-первых, это позволяет разграничивать загрузочные, системные и пользовательские файлы, во-вторых, использовать на каждом из разделов свой тип файловой системы, в-третьих — устанавливать на один ПК несколько разных операционных систем.

Существует две основных схемы разбиения на партиции. Самой распространённой является MBR. Называется она так потому, что в первых физических секторах жёсткого диска этого типа размещается особая область, содержащая загрузочный код и таблицу разделов. Эта область ещё именуется главной загрузочной записью (master boot record или сокращённо MBR).



Эта область диска не является ни одним из логических разделов, и она не доступна для просмотра средствами операционной системы. Загрузочный код передаёт управление компьютером системному разделу, а таблица разделов указывает, где именно начинается и заканчивается тот или иной логический раздел. Основной недостаток MBR заключается

в том, что отводимая под него область диска является фиксированной, а это значит, что в него можно записать ограниченное количество информации. В свою очередь это становится следствием других ограничений, а именно:

- На MBR-диске нельзя создать более четырёх логических Primary-разделов (ограничение условно снимается посредством создания extended-партиции).
- Каким бы объёмным ни был диск, пользователю будет доступно только 2 Терабайта.

Кроме того, схема MBR не отличается надёжностью. Малейшее повреждение кода в этой области приведёт к невозможности загрузки или другим проблемам, при которых записанная на диск информация перестанет определяться.



GUID Partition Entries Array – это аналог Partition Table в MBR, содержащий список всех партиций на диске GPT. В отличие от MBR, он не имеет жёсткой фиксации, поэтому на GPT-диске можно создавать практически неограниченное количество логических разделов.

Другим важным отличием GPT-дисков является резервирование загрузочных данных и сведений о таблице разделов. Если в MBR-дисках они хранятся в одном месте — в первых физических секторах, то в дисках с разметкой GPT они могут храниться где-то ещё, но уже в виде копий. Если основные данные окажутся повреждены, механизм GPT восстановит их из бекапа.

Разметка GPT позволяет работать с дисками объёмом больше 2 Тб.

36. Адресация секторов жесткого диска. Типы адресации. Барьеры размеров дисков.

Почему возникли? Какие присутствуют?

ТОЖЕ САМОЕ ЧТО И В 34

СЕТИ ЭВМ

ОТВЕТЫ НА ТЕСТЫ

1. Управление потоком – возможность принимающей стороны ограничить [**объем**] или [**скорость**] передачи данных по каналу связи.
2. Всегда можно найти два кварцевых генератора с абсолютно одинаковой частотой. **Неверно**
3. Какой(ие) из перечисленных способов конфигурации сетевых узлов допускает(ют) централизованное распространение дополнительных параметров?
Автоматическая конфигурация
4. Укажите, какие из приведённых записей IP адресов являются допустимыми?
192.168.1.10, 10.0.0.1/24, 1.1.1.1 255.255.254.0, 0.0.0.0/0
5. Укажите соответствие режимов работы каналов. **Источник и приемник по очереди меняются местами** → **Разделяемый режим, Данный могут одновременно передаваться от источника к приемнику и от приемника к источнику** → **Двунаправленный, Передача данных осуществляется в одну сторону (от источника к приемнику)** → **Однонаправленный режим**
6. Какие из перечисленных адресов являются адресами узлов? **10.0.1.0/16, 192.168.1.1/28, 127.0.0.1/32**
7. Используя среду Cisco Packet Tracer, выполните задание. В случае выполнения задания в полном объеме на экране появится кодовое слово, которое необходимо ввести в поле ответа. **0xFFDDEEA**
8. Какие из перечисленных диапазонов адресов являются зарезервированными для определённых целей? **0.0.0/8, 10.0.0/8, 172.16.0.0/16, 192.168.0.0/16, 169.254.0.0/16**
9. Укажите верно ли приведенное ниже утверждение?"В вычислительной системе Минск-222 использовалась технология коммутации пакетов". **Неверно**
10. Какие параметры кабеля определяет категория витой пары? **количество используемых проводников, частота скручивания пары проводников между собой, полоса частот линии связи**
11. Временное разделение каналов (TDM) – выделение среды передачи данных каналам на [**определенный**] промежуток времени.
12. Как называется совокупность технических устройств и физической среды, обеспечивающая передачу и распространение сигналов от передатчика к приемнику? **линия передачи данных**
13. Какие из ниже перечисленных характеристик влияют на способность линии связи передавать информацию? **длина линии связи, затухание сигнала, волновое сопротивление линии связи, частотные характеристики линии связи**
14. Укажите верно ли следующее утверждение?"В одной телекоммуникационной линии может быть только один канал передачи информации". **Неверно**
15. Используя среду Cisco Packet Tracer, выполните задание. В случае выполнения задания в полном объеме на экране появится кодовое слово, которое необходимо ввести в поле ответа. **4FFED007**
16. Какой из перечисленных уровней модели открытых систем **полностью** отсутствует в стеке протоколов TCP/IP? **физический уровень**
17. Кто из великих императоров в ходе боевых действий использовал способ связи, предложенный в 1792 году Клодом Шаппом? **Наполеон**
18. Кто впервые организовал межконтинентальный телеграф? **Сайрус Филд**
19. Сколько сетевых узлов может иметь сеть 123.123.123.127/25. **126**
20. Какой протокол используется для поиска соответствия IP адреса MAC адресу сетевого узла? **ARP**

21. Какой из перечисленных кодов позволяет безошибочно выявить последовательно идущие 1 и 0? **Манчестер 2**
22. В асинхронном режиме работы телекоммуникационной линии старт-стоп биты используются для **синхронизации** передачи данных...
23. Укажите соответствие классов компьютерных сетей и соответствующих IP... **A (1-126), B(128-191), C(192-223), D(224-239), E(240-254)**
24. Как называются дополнительные поля, содержащие информацию о передаваемых данных? **Контрольная сумма**
25. Какие города впервые соединил оптический телеграф? **Лиль – Париж**
26. В записи 100BaseTX цифра до слова BASE означает **[Скорость линии передачи составляет 100 Мбит/сек]**, а буквы TX - **Скорость линии передачи составляет 100 бод**
27. Укажите соответствие разводки кабеля и его типа: **TIA/EIA-568-B - TIA/EIA-568-В патч-корд, TIA/EIA-568-B - TIA/EIA-568-A кросс-кабель.**
28. Какие из перечисленных технологий считаются классическими способами....
Коммутация каналов, коммутация пакетов.
29. Укажите фамилию изобретателя 1-го оптического телеграфа(семафора). **Шапп**
30. Какие из перечисленных международных организаций и профессиональных сообществ участвуют в разработке и утверждении стандартов в отрасли сетей ЭВМ телекоммуникаций? **ВСЕ ВАРИАНТЫ**
31. Определите соответствие типов скоростей передачи информации и единиц их измерения. **Модуляционная скорость передачи информации – боды.**
Информационная скорость передачи информации – биты в секунду
32. Какие из перечисленных модуляций не используются в сетях ЭВМ и
Сдвиговая модуляция.
33. Определите соответствие телекоммуникационных интерфейсов и классов физичес... **XDSL (ПРОВОДНОЕ), NFC(БЕСПРОВОДНОЕ)**
34. Какова была скорость передачи информации с использованием изобретения Кл....
2 слова в минуту
35. Какие символы использовались в электромагнитном телеграфе Сэмюэлем Морзе....
Точка, тире
36. Какое название своему изобретению высокоскоростной передачи информации на расстоянии дал Клод Шапп? **Семафор**
37. Укажите соответствие способа управления потоком и его описания.
1)...**Передаются порциями (кадрами). Очередная порция передается только после подтверждения получения текущей порции (режим остановка – ожидание).** 2) Передаются до тех пор, пока принимающая сторона не попросит приостановить передачу. **Восстановление передачи осуществляется так же по просьбе принимающей стороны(режим XON/XOFF).** 3)**Очередной кадр передается только при наличии сигнала о готовности к приему (аппаратное управление потоком).** 4) **данные, полученные по каналу, помещаются принимающей стороной во временное хранилище, из которого затем извлекаются и обрабатываются (организация буфера на принимающей стороне)**

/*Удаление лишних пробелов*/

```
#include <stdio.h>
#include <string.h>

int deletes(char *s, int pos)
{ int i;
if ((s[pos]== ' ') && (s[pos+1]== ' '))
{ deletes(s, pos+1); }
for (i = pos; i < strlen(s)-1; i++)
{
    s[i]=s[i+1];
}
s[i]=0;
return 0;
}
int main(int argc, char* argv[])
{
char s[150];
int i;
gets(s);
for (i = 0; i < strlen(s); i++)
{ if ((s[i]==' ') && (s[i+1]==' '))
    deletes(s, i+1);
}
printf("%s", s);
return 0;
}
```

/*Раскладывает число на простые множители */

```
#include <stdio.h>
#include <stdlib.h>
int func(int par, int *a)
{
    int i, index = 0;
    for(i=2;i<=par;)
        if((par%i)==0) {

            a[index ++] = i;
            par/=i;

        } else i++;

    a[index] = 0;//заполняем последний или единственный
                // (если вдруг множители были не обнаружены) элемент нулём

    for(i=0;i<index;i++)
        printf("%d * ", a[i]);

    return 0;
}
int main()
{
    int n , a[100];
    scanf("%d", &n);
    printf("%d = ", n);
    func(n,a);
    return 0;
}
```

/*. Написать функцию, "переворачивающую" в строке все слова.
(Например: "Жили были дед и баба" - "илиЖ илыб дед и абаб").*/

```
#include <stdio.h>
#include <string.h>

int func(char *str1, char *str2){
    int i, j, len, pos1 = 0, pos2 = 0;
    int index = 0;

    len = strlen(str1);      // определяем длину исходной строки, кол-во
    символов
    for(i = 0; i <= len; i++){ // цикл, по всем символам исходной строки
        if ((str1[i] == ' ') && (i == 0)){ //если встретили пробел, в начале
            while(str1[i] == ' '){           //пробегаем все пробелы
                str2[index ++] = str1[i++]; //и записываем их в результирующую
                //строчку, так же как они были
                //в исходной строчки
            }
            pos1 = i;                      //запоминаем позицию
        }
        if (((str1[i] == ' ') && (i != 0)) || (i == len)){ //если встретили
            //пробел
            //не в начале строчке
            //или в конце строчке
            //запоминаем вторую
            позицию
            pos2 = i - 1;
            for(j = pos2;j >= pos1;j --){ //цикл, со второй до первой
                //позиции (т.е. с конца до
                начала слова)
                str2[index ++] = str1[j]; //записываем в результирующую
                //строку слово наоборот
            }
            if (str1[i] == ' ') str2[index ++] = ' ';// если встретили пробел
            //записываем его в
            результирующую
            //строку на ту же позицию как и
            в
            //исходной строке
            pos1 = i + 1;                  //запоминаем позицию
        }
    }
    str2[len] = '\0';          //записываем в конец результирующей строки,
    символ
    printf("%s", str2);        // конца строки
    return 0;
}

int main()
{
char s1[150], s2[150];
gets(s1);
func(s1,s2);
return 0;
}
```

/*Написать функцию, выполняющую поиск в строке два одинаковых фрагмента, не содержащих пробелы и имеющих максимальную длину, и возвращающую указатель на начало первого из них.*/

```
#include <stdio.h>
#include <string.h>

int search(char *text)
{
    int i,j,ii,jj;
    int kol;           //кол-во символов в строке
    int pos,pos_max;
    int sim,sim_max;
    kol = strlen(text);      // определяем длину исходной строки, кол-во символов
    sim_max=-1;
    pos=-1;
    for(i=0;i<kol;i++) {           //цикл по всем символам в строки
        for(j=i+1;j<kol;j++) {     //от текущего символа до конца строки
            if(text[i]==text[j]) {  //сравниваем первые символы предполагаемых фрагментов
                ii=i;jj=j;
                pos=ii; // запоминаем позицию первого фрагмента
                sim=0;
                while(text[ii]==text[jj]) { //цикл определяет max длину фрагментов
                    if((text[ii]==' ')||(text[jj]==' '))break; //идём до пробела
                    sim++;           //длинна цепочек
                    if(ii<kol) ii++;else break; //пока не конец строки
                    if(jj<kol) jj++;else break; //пока не конец строки
                }
                if(sim>sim_max){sim_max=sim;pos_max=pos;} //если эти фрагменты больше
            }                               //запоминаем их
        } }
    printf("Position %d, symbols %d \n",pos_max,sim_max);
    return pos_max; // возвращаем позицию первого фрагмента
}
int main()
{
char s1[150];
gets(s1);
search(s1);
return 0;
}
```

```

/*Написать функцию, выполняющую поиск в строке конец
предложения, обозначенный символом "точка",
и заменяющую в следующем слове первую строчную на
прописную букву. Между словами количество пробелов может
быть любым.*/

```

```

#include <stdio.h>
#include <string.h>
#define CHECK(alf) (((alf >= 'a') && (alf <= 'z')) || ((alf >= 'A') && (alf
<= 'Z')))

int func(char *str1, char *str2){
    int i;
    int len;

    len = strlen(str1);           //определяем длину заданной строки, т.е.
количество символов в строке.
    for(i = 0;i < len;i ++){      //цикл по всем символам в строке
        if (str1[i] == '.'){      //Если встретили точку
            str2[i] = str1[i];   //записываем ее в результирующую строку
            i++;                 //переходим к следующему символу
            if (str1[i] == ' ')     //если в заданной строке встретили
пробел
                str2[i] = str1[i]; //записываем его в результирующую строку

            while(!CHECK(str1[i])) && (str1[i] != '\0')){ // если после точке,
встречаем не букву или
                //символ конца строки
            i++;
            str2[i] = str1[i];   //записываем этот символ в результирующую
//строку
        }
    }

    if ((str1[i] >= 'a') && (str1[i] <= 'z')) str2[i] = str1[i] - 32; //если
после точке встречаем строчную
    // букву английского языка, заменяем ее на
    // прописную

} else str2[i] = str1[i]; //записываем символы в результирующую строку
}
printf("%s", str2);
return 0;
}
int main()
{
char s1[150], s2[150];
gets(s1);
func(s1,s2);
return 0;
}

```

**/*Разработать подпрограмму на языке Си, заменяющую в строке
принятое в Си обозначение символа с заданным кодом в восьмеричной
системе счисления на сам символ в базовой кодировке ASCII.*/**

```
#include <stdio.h>
#include <string.h>
#define CHECK(c) ((c >= '0') && (c <= '7'))
int func(char *str1, char *str2){
    int i1, i2;
    int len;
    int d, c;

    len = strlen(str1);      //определяем длину заданной строки, т.е.
    количество символов в строке.
    i2 = 0;
    for(i1 = 0;i1 < len;i1 ++){ //цикл, по всей длине строки

        i1++;
        if (CHECK(str1[i1])){ //если после слеша встретили цифру
            d = 0;
            while(CHECK(str1[i1])){
                d = d * 10 + (str1[i1] - 48); //преобразуем набор символов
        в число
                i1++;
            }

            d = make_int(d);           //вызываем функцию преобразования из
            восьмеричной системы
            //счисления в десятеричную
            str2[i2 ++] = d;
            str2[i2 ++] = str1[i1];
        } else {str2[i2 ++] = str1[i1 - 1];str2[i2 ++] = str1[i1];} //если
        после «\» идет не цифра, а любой другой

    }
    str2[i2] = '\0';// записываем в конец результирующей строки символ
    конца строки
    printf("%s", str2);
    return 0;
}

/*функция преобразующая число из восьмеричной системы счисления в
десятеричную*/
int make_int(int d){ //в функцию передается число в восьмеричной системе
    счисления
    int tmp = 0;
    int c, w = 1;
    while(d != 0){ // делаем преобразования пока число не станет равным
        нулю
        c = d % 10; //выделяем цифру и умножаем на w, а w - это степень
        восьмерки
        tmp += c * w;
        w *= 8;
        d /= 10;
    }
    return tmp; //возвращаем преобразованное число
```

```
}

int main()
{
char s1[150], s2[150];
gets(s1);
func(s1,s2);
return 0;
}
```

/*Разработать подпрограмму на языке Си, выполняющую поиск слова в строке, начинающегося с самой младшей латинской буквы, и возвращающую указатель на его начало.*/

```
#include <stdio.h>
#include <string.h>
#define CHECK(c) (((c >= 'a') && (c <= 'z')) || ((c >= 'A') && (c <= 'Z')))
#define MAL(c) ((c >='a') && (c<='z'))
char* func(char *str1){
    int i, len;
    char min, tmp;
    char *str2;

    len = strlen(str1); // определяем длину заданной строки, т.е. количество символов
    while((!CHECK(*str1)) && (*str1 != '\0')) // пропускаем все символы не являющиеся буквами латинского
        str1++; // алфавита

    min = (MAL(*str1)) ? (*str1 - 32) : *str1; //встретили первую латинскую букву, запоминаем код прописной
    // буквы, так как в предложении могут быть как строчные так и // прописные буквы, для удобства сравнения пользуемся кодами
    // прописных букв
    str2 = str1;

    for (;*str1 != '\0';str1++){ //цикл, до конца строчке
        if (*str1 == ' ') { //если встретили пробел
            while((!CHECK(*str1)) && (*str1 != '\0')) //а за пробелом следует символ, не являющийся буквой
                str1++; // латинского алфавита, пропускаем его

            if (*str1 != '\0'){ //если не дошли до конца строки
                tmp = (MAL(*str1)) ? (*str1 - 32) : *str1; //сравниваем текущую букву с ранее записанной
                if (min >= tmp) { //если она младшей или равна, то запоминаем ее
                    //как минимум
                    min = tmp;
                    str2 = str1;
                }
            }
        }
        printf("%s\n",str2);
    return str2; //возвращаем указатель на начало слова, начинающегося с самой младшей латинской буквы
}
int main()
{
char s1[150];
gets(s1);
func(s1);
return 0;
}
```

```

/*Разработать подпрограмму на языке Си, удаляющую из
строки комментарии вида /* ... */
/*Игнорировать вложенные комментарии.*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* comment (char * fname){
    FILE *in;
    char input;
    int stat;
    char *text;
    int kol=0;
    in=fopen(fname,"r"); //открываем файл
    stat=0;
    text=calloc(1,sizeof(char)); //выделение памяти для выходной
строки
    while(fscanf(in,"%c",&input)!=EOF) {
        if(input=='/'){ // если встретили «/»
            fscanf(in,"%c",&input); // считываем из файла один
СИМВОЛ
            if(input=='*') stat++; //и встретили «*» после «/»,
поднимаем флаг
        }
        if((input=='*')&&(stat>0))
{
            fscanf(in,"%c",&input); // считываем из файла один символ
            if(input=='/') {stat--}; //если встретили “*/” опускаем флаг
            fscanf(in,"%c",&input); // считываем из файла один символ
}
        if(stat==0){
            text = realloc(text, (kol+2) * sizeof(char)); //выделяем
память для
            text[kol]=input; // следующего элемента
            //заносим текущий символ в выходную
строку
            kol++;
        }
    }
    fclose(in);
    printf("%s", text);
return text; //возвращаем указатель на начало результирующей строки
}
int main()
{
char s1[150];
gets(s1);
comment(s1);
return 0;
}

```

/*Разработать подпрограмму на языке Си, выполняющую поиск в строке наиболее часто встречающегося символа, и заменяющую его на пробел.*/

```
#include <stdio.h>
#include <string.h>

int func(char *str1, char *str2){
    int i, j, len;
    int max = 0, tmp_max = 0;
    char c;

    len = strlen(str1); // определяем длину строки, то есть количество символов в строке
    for(i = 0;i < len;i ++){ //цикл по всей длине строки
        tmp_max = 0; //переменная для подсчета символов
        if (str1[i] == ' ') continue; // если встретили пробел, пропускаем его, программа продолжается дальше.
        for(j = 0;j < len;j ++){ //цикл, по всей длине строки
            if (str1[i] == str1[j]) tmp_max++; //если нашли одинаковые символы, увеличиваем счетчик
        }
        if (max < tmp_max) {max = tmp_max; c = str1[i];} //если максимальное, уже существующее значение

        // меньше получившегося, то текущее становится максимальным и запоминаем при каком символе
    }

    for(i = 0;i < len;i ++){ //цикл по всей длине строки
        if (str1[i] == c) str2[i] = ' '; //если текущий символ равен чаще всего встречающемуся, заменяем его на пробел
        else str2[i] = str1[i]; //если нет, просто записываем его в результирующую строку как есть
    }
    str2[i] = '\0';
    printf("%s\n",str2); //дописываем в конец
    результирующей строки признак конца строки
    return 0;
}

int main()
{
char s1[150], s2[150];
gets(s1);
func(s1,s2);
return 0;
}
```

/* . Написать функцию, выполняющую поиск подстроки в строке. Стока и подстрока заданы в массивах символов. Результат функции – указатель на начало подстроки в строке или NULL.*/

```
#include <stdio.h>
#include <string.h>
/*: В функцию передаем строку(виде char массива), строку поиска, и их длины*/
int func(char *array,char * search,int kol,int kol_temp)
{
    int pos_str;
    int j=0;
    int start=0;           //флаг статуса (поднимается если найдено начало
искомой подстроки)
    int pos=-1, i;
    for(i=0;i<kol;i++){   //цикл перебирает все символы в строке
        if(start==1){     //если первый символ совпал
            if(j!=kol_temp){ //проверяем дошли или нет до
                //конца искомой подстроки
                if(array[i]==search[j]){j++;}else //считаем кол-во
                    //совпавших символов
                {start=0; j=0;i=pos; pos=-1;} //иначе сбрасываем
флаг
            }else break;
        }
        if((start==0)&&(array[i]==search[0])){start=1;j++;pos=i;} //если
первый
                                    // символ совпадает с текущим поднимаем
флаг
    }
    printf("\nPos is %d",pos+1);
    return pos; //возвращаем позицию найденной последовательности.
}
int main()
{
char s1[150], s2[150];
int n1,n2;
gets(s1);
gets(s2);
scanf("%d %d", &n1, &n2);
func(s1,s2,n1,n2);
return 0;
}
```

/*Написать функцию, определяющую в массиве максимальную длину последовательности расположенных подряд возрастающих значений, и возвращающую индекс ее начала.*/

```
#include <stdio.h>
#define n 10
/*: В функцию передаётся указатель на начало массива и кол-во элементов в этом массиве. */
int search(int *array,int kol){
    int i;
    int cycle;
    int pos,el,el_max,pos_max;
    cycle=0;
    el_max=-1;
    pos_max=0;
    for(i=1;i<kol;i++){
        if(cycle==0){
            if(array[i]>array[i-1]) {cycle=1;pos=i-1;el=2;}
        }else{
            if(array[i]>array[i-1]) {el++;}
            else{
                cycle=0;
                if(el>el_max) {el_max=el; pos_max=pos; }
            }
        }
        if(el>el_max){el_max=el; pos_max=pos; }
        printf("\nMaximalnaya posledovatelnost is %d elementov nachalo %d",el_max,pos_max+1);
    }
    return 0;
}
int main()
{
    int i;
    int arr[n];
    for(i =0 ;i < n; i++)
        scanf("%d", &arr[i]);
    search(arr,n);
    return 0;
}
```

/* . Написать функцию, преобразовывающую дробную часть переменной типа double в строку символов (путем последовательного умножения дробной части на 10). */

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

int main()
{
    char input_str[80];
    double number;
    int i;
    fgets(input_str, 80, stdin);
    int number_of_digits = strlen(input_str) - (strchr(input_str, '.') -
input_str + 1) - 1;
    printf("n = %d\n", number_of_digits);
    sscanf(input_str, "%lf", &number);
    number -= floor(number);
    int digit;
    for( i = 0; i < number_of_digits ; i++)
        printf("%d\n", ((int)(number * pow(10., i + 1))) % 10);

    return 0;
}
```

*/*Разработать подпрограмму на языке Си,
выполняющую поиск наименьшего общего кратного для всех элементов
массива - минимальное число, которое делится на все элементы
массива без остатка.*/*

```
#include <stdio.h>
#define n 5
int krat(int *array,int kol)
{
int rez=-1,i,j;
    for(i=1;i<65535;i++) //перебираем НОК от 1 до 65535
    {
        for(j=0;j<kol;j++)          //смотрим делится ли текущий нок
                                       //на все элементы массива
            if(i%array[j]!=0)break;   //проверяем делится ли НОК
                                       //на текущий элемент массива
        if(j==kol){rez=i; break;}   //если кол-во делящихся
                                       //Эл. Массива = кол-во Эл. Массива НОК
                                       // присваиваем НОК = i
    }
    printf("%d", rez);
    return rez; // возвращаем найденный НОК или -1.
}
int main()
{
int i;
int arr[n];
for(i =0 ;i < n; i++)
scanf("%d", &arr[i]);
krat(arr,n);
return 0;
}
```

**/*Разработать подпрограмму на языке Си, формирующую массив простых чисел в диапазоне от 2 до заданного.
Очередное простое число определяется попыткой деления нацело числа на все уже накопленные простые числа*/**

```
#include <stdio.h>
#include <stdlib.h>
#define n 20
/* в функцию передается переменная (num) и указатель на начало результирующего массива (*mas) */
int func(int num, int *mas){
    int len = 2;
    int i, j, index = 0, flag;
    if (num == 2) {mas[0] = 2; return 1;} //если заданное число равно 2, то оно записывается в массив и программа завершается
    if (num < 2){ printf("ERROR \n"); return 0;} //если число меньше двух, то программа завершается с сообщением об ошибке
    mas[index ++] = 2; //записываем в массив
    первое простое число =2
    for (i = 3;i < num;i ++){ //организуем цикл от 3 до заданного
        числа
        flag = 1;
        for (j = 0;j < len - 1;j ++){ //цикл, по всем элементам массива
            if ((i % mas[j]) == 0) flag = 0; //если число (от 3 до заданного)
            делится на элементы массива на цело
            //устанавливаем flag = 0
        }
        if (flag == 1){ //если число (от 3 до заданного)
            является простым и его нет в массиве
            len ++; //увеличиваем размер массива на
1
            mas[index ++] = i; //и записываем туда это число
        }
    }
    for(i=0;i<len-1;i++)
        printf("%d ", mas[i]);
    return len;
}
int main()
{
    int i;
    int arr[100];
    func(n,arr);
    return 0;
}
```

**/*Разработать подпрограмму на языке Си, осуществляющую двоичный поиск строки по образцу в упорядоченном массиве строк.
Если найдено несколько строк, то требуется найти первую. Функция возвращает указатель на найденную строку.*/**

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>

#define STRCOUNT 5
#define STRLENGTH 10

char rnd_letter(void)
{
    char start = 'a';
    char end   = 'z';
    return rand() % (end - start + 1) + start;
}

int cmp(const void * e1, const void * e2)
{
    return strcmp(*(char **)e1, *(char **)e2);
}

int bin_search(char * strings[STRCOUNT], char * search_str)
{
    int mid, start = 0, end = STRCOUNT - 1;
    int result;
    while(end - start > 1)
    {
        mid = (end - start) / 2 + start;
        result = strcmp(search_str, strings[mid]);
        if(result == 0) return mid;
        if(result < 0)
            end = mid;
        else
            start = mid;
    }
    if(!strcmp(strings[end], search_str)) return end;
    if(!strcmp(strings[start], search_str)) return start;
    return -1;
}

int main()
{
    srand(time(NULL));
    char strings[STRCOUNT] [STRLENGTH];
    char input_str[STRLENGTH];
    char * sorted[STRCOUNT];
    int i, j, clength;
    for(i = 0; i < STRCOUNT; i++)
    {
        clength = rand() % STRLENGTH;
        for(j = 0; j < clength; j++)
            strings[i][j] = rnd_letter();
```

```
        strings[i][j] = '\0';
        sorted[i] = strings[i];
    }

qsort(sorted, STRCOUNT, sizeof(char *), cmp);
printf("Strings:\n");
for(i = 0; i < STRCOUNT; i++)
    printf("%d: \"%s\"\n", i + 1, sorted[i]);

while(1)
{
    printf("\n\nEnter a string to search or 'exit' for termination: ");
    memset(input_str, '\0', STRLENGTH);
    fgets(input_str, STRLENGTH, stdin);
    input_str[strchr(input_str, '\n') - input_str] = '\0';
    if(!strcmp("exit", input_str)) break;
    int result = bin_search(sorted, input_str);
    if(result >= 0)
        printf("Found your at %d position\n", result + 1);
    else
        printf("Can't find your string\n");
}
return 0;
}
```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void BubbleSort(int *arr, int n) {
    int i, j, tmp;

    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            if(arr[j] > arr[j + 1]){
                tmp = arr[j + 1];
                arr[j + 1] = arr[j];
                arr[j] = tmp;
            }
}

int main(int argc, char *argv[]){
    int *arr;
    int i, size = 10;
    // allocation memory
    arr = malloc(sizeof(int) * size);
    srand(time(0));

    // init array (random values 1-100)
    for (i = 0; i < size; i++)
        arr[i] = rand() % 100 + 1;

    printf("start array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n\n");

    BubbleSort(arr, size);

    printf("sorted array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);

    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void BucketSort(int arr[], int size){
    int i, j;
    // find max array element
    int maxValue = arr[0];

    for (i = 1; i < size; i++)
        if (arr[i] > maxValue)
            maxValue = arr[i];

```

```

// create supporting array
int *bucket = malloc(sizeof(int) * (maxValue + 1));
// init bucket array zero values
for(i = 0; i < maxValue + 1; i++)
    bucket[i] = 0;

// distributing values to buckets
for(i = 0; i < size; i++)
    bucket[arr[i]]++;

// countingSort use for sort Bucket values
for(i = 0, j = 0; i < maxValue + 1; i++)
    for(; bucket[i] > 0; (bucket[i])--)
        arr[j++] = i;
}

int main(int argc, char *argv[]){
    int *arr;
    int i, size = 10;
    // allocation memory
    arr = malloc(sizeof(int) * size);
    srand(time(0));
    // init array (random values 1-100)
    for (i = 0; i < size; i++)
        arr[i] = rand() % 100 + 1;

    printf("start array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n\n");

    BucketSort(arr, size);

    printf("sorted array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);

    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void CountingSort(int *arr, int n) {
    int i, j, b = 0;
    int *c;
    int k = 100; // arr values [1..100]

    c = malloc(sizeof(int) * k);

    for(i = 0; i < k; i++)

```

```

c[i] = 0;

for(i = 0; i < n; i++)
    c[arr[i]]++;

for(j = 0; j < k; j++)
    for(i = 0; i < c[j]; i++) {
        arr[b] = j;
        b++;
    }
}

int main(int argc, char *argv[]){
    int *arr;
    int i, size = 10;
    // allocation memory
    arr = malloc(sizeof(int) * size);
    srand(time(0));
    // init array (random values 1-100)
    for (i = 0; i < size; i++)
        arr[i] = rand() % 100 + 1;

    printf("start array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n\n");

    CountingSort(arr, size);

    printf("sorted array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);

    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
    int i, n, p;
    printf("Input n:\n> ");
    scanf("%d", &n);

    int *a = malloc(sizeof(int) * (n + 1));

    for(i = 0; i < n + 1; i++)
        a[i] = i;

    printf("\nPrime numbers in interval [0..%d]:\n", n);
    for(p = 2; p < n + 1; p++)
        if(a[p] != 0){
            printf("%d\n", a[p]);

```

```

        for(i = p * p; i < n + 1; i += p)
            a[i] = 0;
    }

    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// sift through the heap - form heap
void siftDown(int *arr, int root, int bottom){
    int maxChild; // index maximal child of heap
    int done = 0; // is heap formed ? ( yes(1) or no(0) )
    int tmp;

    // while don't go to last row
    while ((root * 2 <= bottom) && (!done)){
        if (root * 2 == bottom) // if there last row
            maxChild = root * 2; // save left child

        // else save larger child of two
        else if (arr[root * 2] > arr[root * 2 + 1])
            maxChild = root * 2;
        else
            maxChild = root * 2 + 1;

        // if root lower than maxChild
        if (arr[root] < arr[maxChild]){
            tmp = arr[root]; // change them
            arr[root] = arr[maxChild];
            arr[maxChild] = tmp;
            root = maxChild;
        }
        else
            done = 1; // pyramid was formed
    }
}

// HeapSort algorithm
void HeapSort(int *arr, int size){
    int i, tmp;

    // form the bottom row of the pyramid
    for (i = (size / 2) - 1; i >= 0; i--)
        siftDown(arr, i, size - 1);

    // sift through the pyramid other values
    for (i = size - 1; i >= 1; i--){
        tmp = arr[0];
        arr[0] = arr[i];
        arr[i] = tmp;
    }
}

```

```

        siftDown(arr, 0, i - 1);
    }

}

int main(int argc, char *argv[]){
    int *arr;
    int i, size = 10;

    // allocation memory
    arr = malloc(sizeof(int) * size);

    srand(time(0));

    // init array (random values 1-100)
    for (i = 0; i < size; i++)
        arr[i] = rand() % 100 + 1;

    printf("start array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n\n");

    HeapSort(arr, size);

    printf("sorted array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);

    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void InsertionSort(int *arr, int n) {
    int i, j;
    int tmp;

    for(j = 1; j < n; j++) {
        tmp = arr[j];
        i = j - 1;

        while(i >= 0 && arr[i] > tmp) {
            arr[i + 1] = arr[i];
            i--;
        }

        arr[i + 1] = tmp;
    }
}

int main(int argc, char *argv[]){

```

```

int *arr;
int i, size = 10;

// allocation memory
arr = malloc(sizeof(int) * size);

srand(time(0));

// init array (random values 1-100)
for (i = 0; i < size; i++)
    arr[i] = rand() % 100 + 1;

printf("start array:\n");
for(i = 0; i < size; i++)
    printf("%d ", arr[i]);
printf("\n\n");

InsertionSort(arr, size);

printf("sorted array:\n");
for(i = 0; i < size; i++)
    printf("%d ", arr[i]);

return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Merges two subarrays of arr[]
// First subarray is arr[0..middle]
// Second subarray is arr[middle + 1..right]
void Merge(int arr[], int left, int middle, int right){
    int i, j, k;

    int n1 = middle - left + 1;
    int n2 = right - middle;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[middle + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0;          // Initial index of first subarray
    j = 0;          // Initial index of second subarray
    k = left;      // Initial index of merged subarray
    while (i < n1 && j < n2) {

```

```

        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else{
            arr[k] = R[j];
            j++;
        }
        k++;
    }

/* Copy the remaining elements of L[], if there are any */
while (i < n1){
    arr[k] = L[i];
    i++;
    k++;
}

/* Copy the remaining elements of R[], if there are any */
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}

// recursive algorithm
void MergeSort(int arr[], int left, int right){
    if (left < right){

        // middle element
        int middle = left + (right - left) / 2;

        // Sort first and second sub-arrays
        MergeSort(arr, left, middle);
        MergeSort(arr, middle + 1, right);

        Merge(arr, left, middle, right);
    }
}

int main(int argc, char *argv[]){
    int *arr;
    int i, size = 10;
    // allocation memory
    arr = malloc(sizeof(int) * size);
    srand(time(0));
    // init array (random values 1-100)
    for (i = 0; i < size; i++)
        arr[i] = rand() % 100 + 1;
    printf("start array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n\n");
}

```

```
MergeSort(arr, 0, size);

printf("sorted array:\n");
for(i = 0; i < size; i++)
    printf("%d ", arr[i]);

return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void OddEvenSort(int *arr, int n) {
    int sorted = 0, i, temp;
    while (sorted != 1) {
        sorted = 1;
        for (i = 1; i < n - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr, i, i + 1);
                sorted = 0;
            }
        }
        for (i = 0; i < n - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr, i, i + 1);
                sorted = 0;
            }
        }
    }
}

// change i and j elements
void swap(int *arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

int main(int argc, char *argv[]){
    int *arr;
    int i, size = 10;

    // allocation memory
    arr = malloc(sizeof(int) * size);

    srand(time(0));

    // init array (random values 1-100)
    for (i = 0; i < size; i++)
        arr[i] = rand() % 100 + 1;

    printf("start array:\n");
```

```

for(i = 0; i < size; i++)
    printf("%d ", arr[i]);
printf("\n\n");

OddEvenSort(arr, size);

printf("sorted array:\n");
for(i = 0; i < size; i++)
    printf("%d ", arr[i]);

return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void QuickSort(int *arr, int l, int r) {
    int i, j;
    int x, buf;
    i = l;
    j = r;
    x = arr[(l + r) / 2];
    do {
        while (arr[i] < x)
            i++;
        while (x < arr[j])
            j--;
        if (i <= j) {
            buf = arr[i];
            arr[i] = arr[j];
            arr[j] = buf;
            i++;
            j--;
        }
    }
    while (i <= j);
    if (l < j)
        QuickSort(arr, l, j);
    if (r > i)
        QuickSort(arr, i, r);
}

int main(int argc, char *argv[]){
    int *arr;
    int i, size = 10;

    // allocation memory
    arr = malloc(sizeof(int) * size);

    srand(time(0));

    // init array (random values 1-100)

```

```

for (i = 0; i < size; i++)
    arr[i] = rand() % 100 + 1;

printf("start array:\n");
for(i = 0; i < size; i++)
    printf("%d ", arr[i]);
printf("\n\n");

QuickSort(arr, 0, size);

printf("sorted array:\n");
for(i = 0; i < size; i++)
    printf("%d ", arr[i]);

return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void RadixSort(int *arr, int n) {
    int i;
    int semiSorted[n];
    int significantDigit = 1;
    int largestNum = findLargestNum(arr, n);

    while (largestNum / significantDigit > 0) { // Loop until we reach the
        largest significant digit
        int bucket[10] = { 0 }; // Counts the number of "keys" or digits that
        will go into each bucket
        for (i = 0; i < n; i++)
            bucket[(arr[i] / significantDigit) % 10]++;
        /*
         * Add the count of the previous buckets,
         * Acquires the indexes after the end of each bucket location in the array
         * Works similar to the count sort algorithm
         */
        for (i = 1; i < 10; i++)
            bucket[i] += bucket[i - 1];
        // Use the bucket to fill a "semiSorted" array
        for (i = n - 1; i >= 0; i--)
            semiSorted[--bucket[(arr[i] / significantDigit) % 10]] = arr[i];
        for (i = 0; i < n; i++)
            arr[i] = semiSorted[i];
        significantDigit *= 10; // Move to next significant digit
    }
}

int findLargestNum(int *arr, int n) {
    int i;
    int largestNum = -1;
    for (i = 0; i < n; i++) {

```

```

        if (arr[i] > largestNum)
            largestNum = arr[i];
    }
    return largestNum;
}

int main(int argc, char *argv[]){
    int *arr;
    int i, size = 10;

    // allocation memory
    arr = malloc(sizeof(int) * size);

    srand(time(0));

    // init array (random values 1-100)
    for (i = 0; i < size; i++)
        arr[i] = rand() % 100 + 1;

    printf("start array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n\n");

    RadixSort(arr, size);

    printf("sorted array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);

    return 0;
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void SelectionSort(int *arr, int n) {
    int i, j;
    int min, tmp;

    for (i = 0; i < n - 1; i++) {
        min = i;

        // find minimal element index
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min])
                min = j;
        }

        // swap with i-element
        tmp = arr[i];
        arr[i] = arr[min];
        arr[min] = tmp;
    }
}

```

```
}

int main(int argc, char *argv[]) {
    int *arr;
    int i, size = 10;

    // allocation memory
    arr = malloc(sizeof(int) * size);

    srand(time(0));

    // init array (random values 1-100)
    for (i = 0; i < size; i++)
        arr[i] = rand() % 100 + 1;

    printf("start array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n\n");

    SelectionSort(arr, size);

    printf("sorted array:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

1) Определите соответствие телекоммуникационных интерфейсов и классов физических соединений

XDSL – проводное

NFC – беспроводное

2) Укажите соответствие уровней модели OSI/ISO их номерам (1 – самый нижний)

Сетевой уровень – 3 уровень

Физический уровень – 2 уровень

Транспортный уровень – 4 уровень

Пользовательский уровень – 7 уровень

Представительский уровень – 6 уровень

Сеансовый уровень – 5 уровень

Канальный уровень – 2 уровень

3) Соответствие разводки кабеля и его типа

TIA/EIA-568-B – TIA/EIA-568-B – патч-корд

TIA/EIA-568-B – TIA/EIA-568-A – кросс-кабель(кроссовер)

4) Соответствие типов скоростей передачи информации и единиц измерения

Модуляционная скорость передачи информации – боды

Информационная скорость передачи информации – биты в секунду

5) Какой из перечисленных уровней модели открытых систем полностью отсутствует в стеке протоколов TCP/IP

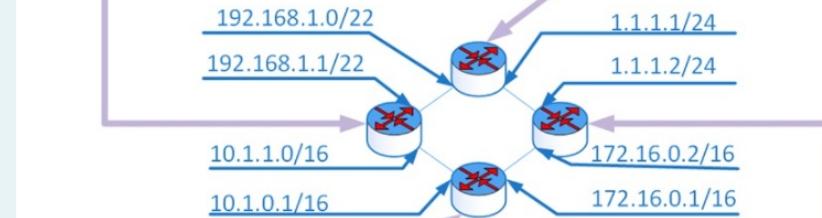
Физический, сеансовый, представительный

6) Понятие «канал передачи». Режимы работы канала.

Вопрос 7
Ответ сохранен
Балл: 1,00
 Отметить вопрос

Какие из IP адресов, указанных на схеме ниже, являются ошибочными?

192.168.0.0 255.255.252.0 0.0.0.0	1.1.1.0 255.255.255.0 10.1.0.1	10.1.0.0 255.255.0.0 0.0.0.0	172.16.0.0 255.255.0.0 192.168.1.0
192.168.0.0 255.255.252.0 0.0.0.0	1.1.1.0 255.255.255.0 0.0.0.0	10.1.0.0 255.255.0.0 192.168.1.1	172.16.0.0 255.255.0.0 1.1.1.2
192.168.0.0 255.255.252.0 172.16.0.1 172.16.0.0 255.255.0.0 0.0.0.0	1.1.1.0 255.255.255.0 0.0.0.0	10.1.0.0 255.255.0.0 1.1.1.1	172.16.0.0 255.255.0.0 0.0.0.0



192.168.0.0 255.255.252.0 10.1.1.0	1.1.1.0 255.255.255.0 10.1.1.0	10.1.0.0 255.255.0.0 0.0.0.0	172.16.0.0 255.255.0.0 0.0.0.0
192.168.0.0 255.255.252.0 172.16.0.1 172.16.0.0 255.255.0.0 0.0.0.0	1.1.1.0 255.255.255.0 0.0.0.0	10.1.0.0 255.255.0.0 1.1.1.1	172.16.0.0 255.255.0.0 0.0.0.0

- 172.16.0.2/16
- 172.16.0.1/16
- 10.1.1.0/16
- 1.1.1.1/16
- 1.1.1.2/16
- Все IP адреса указаны верно
- 192.168.1.0/22
- 10.1.1.1/16
- 192.168.1.1/22

Ответ все адреса указаны верно

7) Какой из представленных диапазонов IP адресов(v4) используется для автоматической конфигурации без DHCP сервера

169.254.0.0/16

8) Какие из перечисленных диапазонов адресов являются зарезервированными для определенных целей

169.254.0.0/16

10.0.0.0/8

192.168.0.0/16

0.0.0.0/8

9) Какие параметры кабеля определяет категория витой пары

количество проводников, полоса частот, длина кабеля

10) Какие из перечисленных адресов являются адресами узлов

b, c, (d?)

11) Какова была скорость передачи информации с использованием изобретения Клода Шаппа(семафор)

2 слова в минуту

12) Укажите какие из перечисленных ниже физ явлений человечество на ранних этапах своего развития использовало для передачи информации

Изображения

Огонь

14) В каком режиме работает dns сервер если он пытается найти ответ на полученный запрос у других dns серверов проходя все уровни пространства имен

рекурсивный режим? итеративный? режим промежуточного DNS (DNS forwardng)??

15) Каким символом указываемым в крайнем правом положении dns имени указывается корневой узел пространства имен

точка

16) Какие из ниже перечисленных характеристик влияют на способность линии связи передавать информацию?

затухание сигнала

частотные характеристики линии

длина линии связи

уровень магнитных волн на солнце

волновое сопротивление линии

17)

Вопрос 20

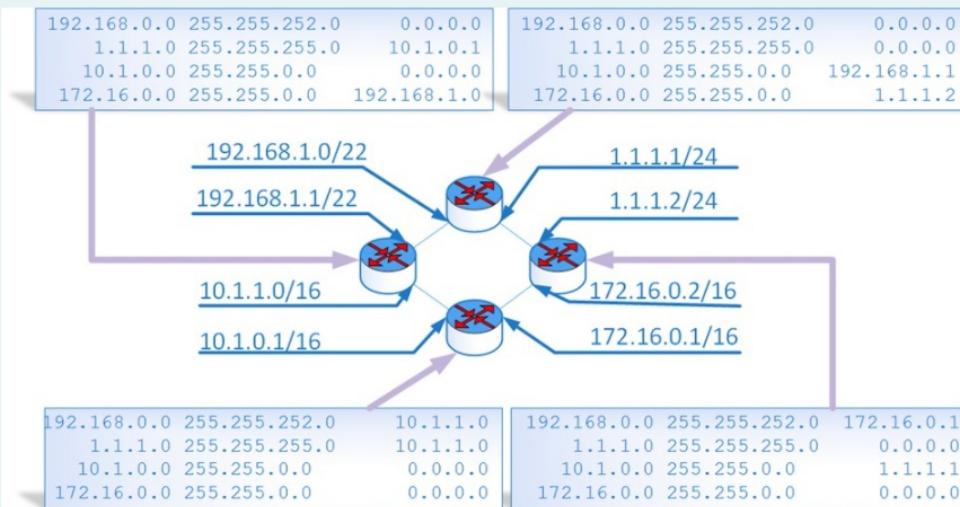
Ответ сохранён

Балл: 1,00

Отметить вопрос

Укажите верно ли следующее утверждение?

Если пакет будет отправлен маршрутизатором, которых располагается внизу схемы, узлу 1.1.1.1, то он будет доставлен по назначению.



Выберите один ответ:

 Верно Неверно**18)** Укажите верно ли следующее утверждение:

1 бод всегда равен 1 бит в секунду

верно

неверно**19)** Какую команду оболочки cmd можно использовать для вывода информации о настройках dns клиента**ipconfig****20)** Какие из перечисленных типов сообщений передаются широковещательно (при первичном диалоге клиента и сервера)**DHCPOFFER, DHCPREQUEST**

21) Сколько сетевых узлов может иметь сеть 123.123.123.127/25?

126

22) Укажите, какие из приведенных записей IP адресов являются допустимыми?

192.168.1.10**10.0.0.0.1/24****23)** Какие из перечисленных условий должны быть выполнены чтобы правильно функционировала система автоматической конфигурации сетевых узлов

ОТВета нет

24) Какие из перечисленных адресов являются адресами узлов?

1.1.1.0/24

192.168.1.1/28

10.0.1.0/16

127.0.0.1/32

10.0.0.0/8

25) Укажите назначение опциональных параметров сетевой конфигурации предоставляемой по протоколу DHCP

3 (Router) - такой опции в DHCP не существует

252 (WPAD) - путь к файлу автоматической конфигурации Proxy-сервера

12 (Host Name) - перечень маршрутизаторов по умолчанию

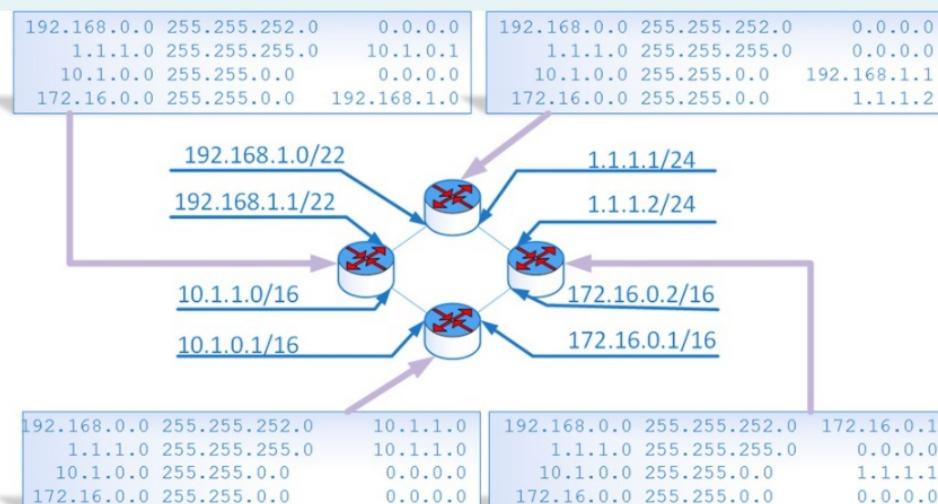
28 (Broadcast address) - адрес широковещательной передачи

26)

Вопрос 20
Ответ сохранен
Балл: 1,00
 Отметить вопрос

Укажите верно ли следующее утверждение?

Если пакет будет отправлен маршрутизатором, которых располагается внизу схемы, узлу 1.1.1.1, то он будет доставлен по назначению.



Выберите один ответ:

- Верно
 Неверно

27) Статическая маршрутизация. Классовая маршрутизация. Бесклассовая

маршрутизация. Технологии CIDR и VLSM.

28) Укажите соответствие классов компьютерный сетей и соответствующих диапазонов IP адресов.

Класс Е 240.0.0.0 - 255.255.255.255

Класс А 0.0.0.0 - 127.255.255.255

Класс С 192.0.0.0 - 223.255.255.255

Класс D 224.0.0.0 - 239.255.255.255

Класс В 128.0.0.0 - 191.255.255.255

29) Какие символы допустимы в именах DNS

латинские символы, знак тире, символ точка, цифры от 0 до 9

31) Какие из перечисленных имен dns содержат информацию о доменах на национальных языках

кафедравс.сибгугти.рф

сибгугти.рф.

xn–90aenc5bjg.xn–p1ai

xn–80aagge2a9bkv.xn–90aenc5bjg.xn–p1ai

32) Какую команду в ОС GNU/Linux можно использовать для определения текущего DNS имени сетевого узла?

nslookup

33) Возможно ли в одном сегменте функционирование нескольких DHCP серверов

Да, при согласованной их конфигурации

34) Укажите пропущенное слово в представленном определении:

... - это часть пространства имен, которая хранится на одном из DNS серверов и изменяется им.

Ответ - слово на русском языке в именительном падеже.

ЗОНА?

35) С какой целью в конфигурации DNS клиента могут быть указаны суффиксы

для указания адресов dns серверов которым клиент должен отправлять поисковые запросы

для определения порядка формирования поисковых запросов

36) Укажите пропущенные слова в определениях типов имен:

абсолютный путь - это путь к объекту именования, указанный начиная с корневого элемента дерева

относительный путь - это путь к объекту именования, указанный относительно другого объекта именования

37) Укажите соответствие отправки пакета при подтверждении аренды сетевых настроек

DHCP сервер отправляет пакет - DHCPOFFER

клиент DHCP отправляет пакет с типом - DHCPREQUEST

38)

Содержит отказ от использования сетевой конфигурации - DHCPDECLINE

Используется для сообщений об освобождении сетевой конфигурации - DHCPRELEASE

Используется для запроса сетевых параметров - DHCPDISCOVER

Содержит предлагаемую сетевую конфигурацию - DHCPIINFORM

Содержит отказ клиента использовать предлагаемую сетевую конфигурацию - DHCPNACK

39) верно ли расшифрована аббревиатура протокола сетевой службы DHCP

DHCP - Dual hydrogen control panel

неверно

40) какие из перечисленных DNS имен являются полностью определенными (FQDN)

sibsutis.ru.

www.ngs.ru

eios.sibsutis.ru

41) какие из ниже перечисленных элементов сетевой инфраструктуры могут присутствовать в сети, в которой функционирует протокол DHCP?

Узел - сервер DHCP

узел со статической сетевой конфигурацией

узел - ретранслятор DHCP

узел - клиент DHCP

42) Могут ли в одном сегменте функционировать одновременно DHCP серверы работающие под управлением операционных систем linux и windows

нет

43) Какова максимальная глубина дерева в пространстве имен DNS (сколько доменов может быть в одном имени)?

127

44) Верно ли следующее утверждение? В случае использования АРРА возможно автоматически конфигурировать сетевые узлы на использование до 5 маршрутизаторов
неверно

45) Составьте текст расшифровывающий аббревиатуру APIPA:

Automatic Private IP Addressing

46) Протокол DHCP допускает аренду сетевых адресов на ограниченное время
верно

47) используя протокол DHCP возможно распространять по сети конфигурацию различных сетевых приложений

верно

48) Какие уровни управления принято выделять при формировании системы имен?

административный уровень

управленческий уровень

глобальный

уровень узлов

всеобщий уровень

49) Какой тип пакета используют DHCP сервера для обмена служебной информации?

DHCPACK

50) Какие из перечисленных сетевых технологий и служб не могут функционировать в условиях использования APIPA

Выберите один или несколько ответов:

Конфигурирование подсетей (Multiple subnets)

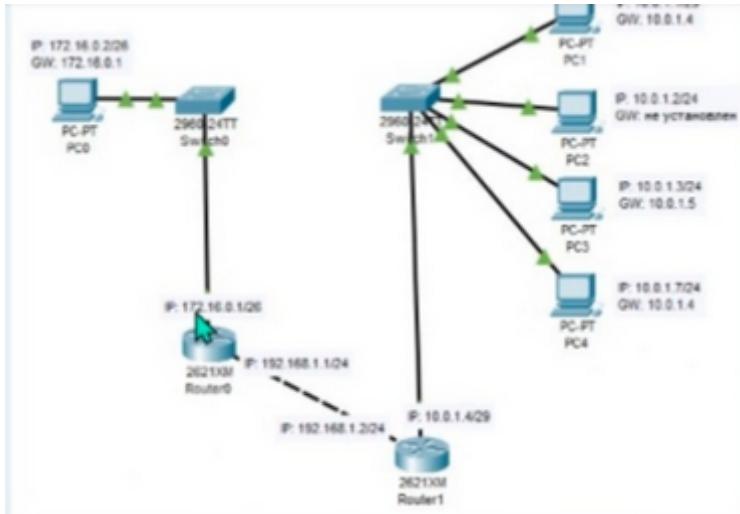
Доменные службы (Active Directory)

51) Вы администратор сети, изображенной выше. Сетевая конфигурация каждого из узлов представлена на рисунке. Между маршрутизаторами настроена и находится в рабочем состоянии динамическая маршрутизация (протокол значения не имеет).

Вы находитесь возле компьютера PC0 (изображен слева сверху), открыли командную строку и набрали команду *ping 10.0.1.4*. Достигнет ли пакет с *ping* цели, придет ли ответ на него и какой адрес отправителя будет указан в ответе, если он придет?

Выберите один ответ:

Пакет ping достигнет назначения, ответ не придет



52) Вы администратор сети, изображенной выше. Сетевая конфигурация каждого из узлов представлена на рисунке. Между маршрутизаторами настроена и находится в рабочем состоянии динамическая маршрутизация (протокол значения не имеет).

Вы находитесь возле компьютера PC4 (изображен справа нижний), открыли командную строку и набрали команду *ping 172.16.0.2*. Достигнет ли пакет с *ping* цели, придет ли ответ на него и какой адрес отправителя будет указан в ответе, если он придет?

Пакет ping достигнет назначения, ответ придет в поле “Адрес отправителя” ответа будет указано 10.0.1.1

53) В ответе необходимо указать Какова максимальная длинна имени узла в пространстве имен DNS?

В ответе необходимо указать количество символов цифрами, например, 76.

Ответ: 63

54) Укажите соответствие изобретателей телеграфов физические явления, которые эти изобретатели использовали для передачи информации.

Клод Шапп оптика(человеческое зрение) галочка

Сэмюэл Морзе оптика электромагнетизм х

Карл Фридрих Гаусс и Вильгельм Вебер электромагнетизм х

Самуэль Томас фон Земмеринг химический эффект электрического тока галочка

55) Как называются дополнительные поля, содержащие информацию передаваемых данных?

Выберите один ответ:

контрольная сумма

никак не называются

мусор

56) Верно ли следующее утверждение?

"При передачи информации по электромагнитному телеграфу Самуэля Морзе для разграничения передаваемых букв не

использовались никакие технические или логические способы".

Выберите один ответ:

Верно

Неверно

57) Вы получили следующий кадр:

0010100011011110110

Используя корректирующий код Хемминга укажите какой разряд полученном кадре испорчен.

PS.Ответ следует указать цифрой.Если ошибки в полученном кадре нет,то следует указать значение 0.

Ответ: 0

58) Какие из ниже перечисленных характеристик влияют на способность линии связи передавать информацию?

Выберите один или несколько ответов:

уровень магнитных волн на солнце

волновое сопротивление линии связи

затухание сигнала

частотные характеристики линии связи

длина линии связи

59) Укажите верно ли следующее утверждение

Всегда можно найти два кварцевых генератора с абсолютно одинаковой частотой

Выберите один ответ:

Верно

Неверно

60) Какое название своему изобретению "высокоскоростной" передачи информации на расстояния дал Клод Шапп?

Ответ: семафор

61)

The screenshot shows a network diagram and several configuration tables. The network diagram consists of four nodes arranged in a square. Arrows point from the top-left node to the top-right and bottom-left nodes, and from the bottom-left node to the bottom-right node. The top-left node is associated with the IP range 192.168.1.0/22. The top-right node is associated with 1.1.1.1/24. The bottom-left node is associated with 192.168.1.1/22. The bottom-right node is associated with 1.1.1.2/24. The configuration tables are as follows:

1.1.1.0 255.255.255.0	10.1.0.1
10.1.0.0 255.255.0.0	0.0.0.0
172.16.0.0 255.255.0.0	192.168.1.0

1.1.1.0 255.255.255.0	0.0.0.
10.1.0.0 255.255.0.0	192.168.1.
172.16.0.0 255.255.0.0	1.1.1.

192.168.0.0 255.255.252.0	10.1.1.0
1.1.1.0 255.255.255.0	10.1.1.0
10.1.0.0 255.255.0.0	0.0.0.0
172.16.0.0 255.255.0.0	0.0.0.0

192.168.0.0 255.255.252.0	172.16.0.
1.1.1.0 255.255.255.0	0.0.
10.1.0.0 255.255.0.0	1.1.
172.16.0.0 255.255.0.0	0.0.0.

Below the tables is a list of options for subnet masks:

- 1.1.1.1/16
- 1.1.1.2/16
- 172.16.0.2/16
- 192.168.1.0/22
- 172.16.0.1/16
- 10.1.1.1/16
- 192.168.1.1/22
- Все IP адреса указаны верно
- 10.1.1.0/16

A message at the bottom left says: "Ваш ответ неверный." (Your answer is incorrect). A red button labeled "Неверно" (Incorrect) is visible. On the right side, there is a sidebar with user profiles for Sergey N. Mam, Alexey Romanu, IV-922 Lunev D, and IV-922 Oshlak.

62)

Вопрос 1
Неверно
Баллов: 0,00 из 1,00
 Отметить вопрос
 Редактировать вопрос

Укажите верно ли следующее утверждение?
Если пакет будет отправлен маршрутизатором, которых располагается внизу схемы, узлу 192.168.1.0, то он будет доставлен по назначению.

Выберите один ответ:

Верно
 Неверно

Правильный ответ: Верно

Оставить комментарий или переопределить балл

63) Вы получили следующий кадр:

00101001110111101101

Используя корректирующий код Хемминга укажите какой разряд полученном кадре испорчен.

PS.Ответ следует указать цифрой.Если ошибки в полученном кадре нет,то следует указать значение 0.

Ответ: 8

64) Вы получили следующий кадр:

010101110111101101

Используя корректирующий код Хемминга укажите какой разряд полученном кадре испорчен.

PS.Ответ следует указать цифрой.Если ошибки в полученном кадре нет,то следует указать значение 0.

Ответ: 8

65)

00 из 100

Укажите верно ли следующее утверждение?

Если пакет будет отправлен маршрутизатором, которых располагается внизу схемы, узлу 1.1.1.1, то он будет доставлен по назначению.

Путь: 192.168.0.0 255.255.252.0 0.0.0.0 → 1.1.1.0 255.255.255.0 10.1.0.1 → 10.1.0.0 255.255.0.0 0.0.0.0 → 172.16.0.0 255.255.0.0 192.168.1.0 → 192.168.1.0/22 → 192.168.1.1/22 → 10.1.1.0/16 → 10.1.0.1/16 → 1.1.1.1/24 → 1.1.1.2/24 → 172.16.0.2/16 → 172.16.0.1/16 → 192.168.0.0 255.255.252.0 172.16.0.1 1.1.1.0 255.255.255.0 0.0.0.0 10.1.0.0 255.255.0.0 1.1.1.1 172.16.0.0 255.255.0.0 0.0.0.0

Выберите один ответ:

Верно Неверно

>

66) Какие параметры кабеля определяет категория витой пары?

Выберите один или несколько ответов:

количество используемых проводников

частота скручивания пары проводников между собой

полоса частот линии связи

цвет проводников

длина кабеля в бухте

67) Какой из перечисленных кодов позволяет безошибочно выявить последовательно идущие 1 и 0?

Выберите один ответ:

NRZ

AMI

Манчестер II

68)

Вопрос 1
Неверно
Баллов: 0,00 из 1,00
[Отметить вопрос](#)
[Редактировать вопрос](#)

Укажите верно ли следующее утверждение?
Если пакет будет отправлен маршрутизатором, которых располагается внизу схемы, узлу 192.168.1.0, то он будет доставлен по назначению.

Выберите один ответ:

Верно
 Неверно ✕

Правильный ответ: Верно

Оставить комментарий или пересчитать балл

69) Вы получили следующий кадр:

0010 100111011110110

Используя корректирующий код Хемминга укажите какой разряд полученном кадре испорчен.

PS.Ответ следует указать цифрой.Если ошибки в полученном кадре нет,то следует указать значение 0.

Ответ:8

70) Аппаратура передачи данных-технические средства ,предназначенные для передачи данных по линии(ям)связи. Формируют каналы передачи данных

71) Как называется совокупность технических устройств физической среды обеспечивающая передачу распространение

сигналов от передатчика приемнику?(ответ-3 слова в именительном падеже)

Ответ: линия передачи данных

72) Укажите соответствие уровней модели OSI/ISO их номерам(1-самый нижний)

физический уровень 1 Уровень

сетевой уровень 3 уровень

пользовательский уровень 7 Уровень

транспортный уровень 4 уровень

канальный уровень 2 уровень

сессионный уровень 5 Уровень

представительский уровень 6 уровень

73) Вы получили следующий кадр:

00101000110111101101

Используя корректирующий код Хемминга укажите какой разряд полученном кадре испорчен.

PS.Ответ следует указать цифрой. Если ошибки в полученном кадре нет, то следует указать значение 0.

Ответ: 0

74) Вы получили следующий кадр:

0010100111011110110

Используя корректирующий код Хемминга укажите какой разряд полученном кадре испорчен

PS.Ответ следует указать цифрой.Если ошибки в полученном кадре нет,то следует указать значение 0.

Ответ: 8

75) Вы получили следующий кадр:

0010101110111101101

Используя корректирующий код Хемминга укажите какой разряд полученном кадре испорчен.

PS.Ответ следует указать цифрой.Если ошибки в полученном кадре нет,то следует указать значение 0.

Ответ:8

76) Вы получили следующий кадр:

0010100111011110110

Используя корректирующий код Хемминга укажите какой разряд полученном кадре испорчен.

PS.Ответ следует указать цифрой.Если ошибки в полученном кадре нет,то следует указать значение 0.

Ответ: 8