# **Оптимизация циклов (loop optimization)**

## **Развертка (loop unrolling)**

Данный вид оптимизации используется при малом теле цикла (с известной длиной). Вместо цикла с n-итерациями, компилятор будет производить код, который просто повторяется n раз.

Метод может повысить производительность, так как устраняет любые инструкции перехода, что является целью многих оптимизаций. Однако, следует учесть, что, если увеличенный цикл не помещается в кэш, то данный способ не целесообразен.

**Пример:**

Количество итераций/размер массива: 10 000

До: (**111 мкс**)

for (int i = 0; i < iN; i++){  
 res \*= a[i];   
}

После: (**48 мкс**)

for (int i = 0; i < iN; i+=3){  
 res1 \*= a[i];   
 res2 \*= a[i+1];  
 res3 \*= a[i+2];  
}  
res = res1 \* res2 \* res3; //вынесли из цикла, так как операция однократная

## **Объединение циклов (loop fusion)**

Метод используется, когда:

1. количество итераций циклов одинаково либо кратно
2. циклы не зависимы по данным, то есть, каждая последующая операция второго цикла **не** зависит от данных первого цикла

Это позволяет уменьшить дополнительные расходы на циклы. Такая оптимизация не всегда выигрывает в производительности, так как тело цикла может «вывалиться» из кэша, и в этом случае лучше применить [разрезание циклов](#_Разрезание_циклов_(loop).

Инициализировать два массива в отдельных циклах, иногда лучше, чем оба массива одновременно в одном.

**Пример:**

Количество итераций/размер массива: 10 000

До: (**267 мкс**)

for (int i = 0; i < iN; i++){

a[i] = b[i] - 5;}  
for(int i = 0; i < iN-1; i++){

d[i] = e[i] \* 3;}

После: (**125 мкс**)

for (int i = 0; i < iN-1; i++){  
 a[i] = b[i] - 5;  
 d[i] = e[i] \* 3;  
}  
a[iN-1] = b[iN-1] - 5;

## **Разрезание циклов (loop distribution)**

Данный вид оптимизации используется, когда большое тело программы не помещается в кэш, начало вытесняет конец и наоборот, в худшей ситуации данные вытесняются в оперативную память (ОП), что существенно замедляет программу.

Важно понять какая часть/процент попадет в кэш и вывести эту часть в отдельный цикл. Это возможно тогда, когда вычисления внутри тела цикла не зависят друг от друга.

Метод помогает удалить последовательные операторы в отдельный цикл и собрать параллелизуемые операторы в другой цикл.

Применение может мешать зависимости по данным, в таком случае лучше применить [объединение циклов](#_Объединение_циклов_(loop).

**Пример:**

**До:**

for (i=0; i < n; i++) {

x[i] = y[i] + z[i]\*w[i]; /\* S1 \*/

a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /\* S2 \*/

y[i] = z[i] - x[i]; /\* S3 \*/

}

**После:**

/\* L1: parallel loop \*/

for (i=0; i < n; i++) {

x[i] = y[i] + z[i]\*w[i]; /\* S1 \*/

y[i] = z[i] - x[i]; /\* S3 \*/

}

/\* L2: sequential loop \*/

for (i=0; i < n; i++) {

a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /\* S2 \*/

}

## **Перестановка вложенных в друг друга циклов (loop interchange)**

Выделяют 2 основные цели:

1. тело самого внутреннего цикла должно находиться в кэш памяти
2. цикл, который находиться внутри, должен быть максимальной длины

Тут мы встречаем противоречие. Важно понять, что даст больший выигрыш.

Больший выигрыш даст «погружение» в кэш память, значит, возможно, вам придется применить [разрезание циклов](#_Разрезание_циклов_(loop).

**Рассмотрим несколько случаев:**

* «Идеальный». Всё помещается в кэш память. В этом случае самое выгодное сделать самый длинный цикл – внутренним, а самый короткий – самым внешним.
* «Лучший». Вся или большая часть данных находится в памяти непрерывно и самым внутренним циклом проходится, помещаясь в кэш память. Такая ситуация может не соответствовать идеи о том, что самый большой цикл должен находиться внутри. Главная задача «сманить» данные в кэш последовательным чтением, чем больше обращений к этим данным, тем больше выигрыш.
* «Огромные циклы». В любом случае данные не попадут в кэш. Внутрь вложенных циклов разумно вставить самый маленький цикл, потому что он работает с одной областью данных, которая уже находится в кэше. Но если у вас так устроены данные, что из кэша ничего смогли извлечь, тогда есть смысл сделать внутренним самым длинный, а внешним самый короткий, чтобы уменьшить накладные расходы на организацию циклов.

Важно: соизмеряйте выигрыш от разных оптимизаций, так как универсального приема не существует.

Метод нужен для того, чтобы помочь компилятору распараллеливать код.

**Пример:**

Количество итераций/размер массива: 200\*200

**До:** (50516 мкс)

for(int i = 0; i < iN; i++){  
for(int j = 0; j < jN; j++){  
for(int k = 0; k < kN; k++){  
 c[i][j] = c[i][j] + a[i][k] \* b[k][j];}}}

**После:** (47038 мкс)

for(int i = 0; i < iN; i++){   
for(int k = 0; k < kN; k++){ //меняем местами, чтобы данные шли последовательно  
for(int j = 0; j < jN; j++){   
  c[i][j] = c[i][j] + a[i][k] \* b[k][j];}}}

К конечному циклу можно применить [объединение](#_Объединение_циклов_(loop).

## **Де-факторизация циклов (Loop De-factorization)**

Метод используется**,** когда можем преобразовать математические формулы, которые будут более «дешёвыми» для вычисления.

Математические справочники:

**1.** Бронштейн И. Н. и др. Справочник по математике.

**2.** Корн Г., Корн Т. Справочник по математике.

**3.** Абрамовиц М., Стиган И. Справочник по специальным функциям с формулами, графиками и таблицами

Используя, выигрываем в быстроте, к примеру, умножение ~ 4 такта, а деление ~ 1.5 десятка тактов.

**Пример:**

Количество итераций/размер массива: 100

До: **(66 мкс)**

cos (x+ y)

sin (x)

a/x   
b/x  
c/x

После: **(37 мкс)**

cos(x)\*cos(y) – sin(x)\*sin(y)

eix-e-ix/2\*i

tmp = 1/x  
a\*tmp  
b\*tmp  
c\*tmp

## **Разбивка на блоки (Loop tiling)**

Если тело цикла сложное, то можно применить эту оптимизацию, и разбить цикл на более маленькие части. Разбиение пространства итерирования цикла приводит к разбиению массива на меньшие блоки, которые помещаются в кэш, что приводит к улучшению использования кэша, снижению количества промахов. Можно применять вместе с [перестановкой циклов](#_Перестановка_вложенных_в), но не во всех случаях.

Данный метод может увеличить накладные расходы на циклы.

**Пример:**

Количество итераций/размер массива: 100\*100

До: **(383 мкс)**

for (i = 0; i < N; i++)

for (j = 0; j < N; j++)

c[i] = c[i] + a[i, j] \* b[j];

После: **(127 мкс)**

for (i = 0; i < N; i += 2)

for (j = 0; j < N; j += 2)

for (ii = i; ii < min(i + 2, N); ii++)

for (jj = j; jj < min(j + 2,N); jj++)

c[ii] = c[ii] + a[ii, jj] \* b[ii];

## 

## **Циклическое преобразование кода (Loop-invariant code motion)**

Применяем, если можно вынести одноразовые вычисления за цикл, чтобы не вычислять их несколько раз. Этот метод значительно повышает производительность.

**Пример:**

Количество итераций/размер массива: 10 000

До: **(125 мкс)**

for (int i = 0; i < n; ++i) {

x = y + z;

a[i] = 6 \* i + x \* x;

}

После: **(113 мкс)**

x = y + z;

t1 = x \* x;

for (int i = 0; i < n; ++i) {

a[i] = 6 \* i + t1;

}

## **Loop unswitching**

Используем, когда условие внутри тела цикла мешает его распараллеливанию. На современных процессорах ради скорости, так как они быстро работают на векторах.

**Пример:**

Количество итераций/размер массива: 100 000

До: **(760 мкс)**

int i, w, x[1000], y[1000];

for (i = 0; i < 1000; i++) {

x[i] += y[i];

if (w)

y[i] = 0;

}

После: **(527 мкс)**

int i, w, x[1000], y[1000];

if (w) {

for (i = 0; i < 1000; i++) {

x[i] += y[i];

y[i] = 0;

}

} else {

for (i = 0; i < 1000; i++) {

x[i] += y[i];

} }

# 

**Индексация циклов к нулю**Важно помнить, что сравнения *i* (счётчик цикла) с произвольной константой нет, можно сравнивать только с 0. Такое преобразование цикла существенного упростит проверку условия цикла, она будет более быстрой.

**Обрыв цикла**

Делайте так мало итераций цикла, как это возможно. Можете закончить – заканчивайте. Однако не стоит чрезмерно усложнять условные переходы.

**Оператор continue**

Существует несколько моментов использования данного оператора:

* Он используется, чтобы не считать лишнего на данной итерации. В этом есть смысл тогда, когда экономите больше 30 итераций, так как continue используется с if и в ином случае это бесполезно
* Он используется, чтобы пропустить недопустимые действия. Это вполне обосновано, однако можно избежать следующими способами: [**разрезать**](#_Разрезание_циклов_(loop) **цикл** на два (первый до этого элемента, второй после), если он один; **найти закономерность** (к примеру, идти с шагом два)

Старайтесь устранить «неизбежный» continue, если не помогает – оставьте всё как есть.

**Способы избежать if/else в циклах**

1. Если считать лишнее дешевле чем if/else – считайте.
2. Можно перенести if/else вверх или вниз по циклу, чтобы он попал в проверку цикла
3. [Разрезать](#_Разрезание_циклов_(loop) тело цикла на несколько независимых друг от друга циклов, каждый из которых свободен от if/else
4. Заменить на тернарную операцию или switch
5. Уменьшить количество if/else и сделать условие до else более вероятным, чем условие else

Данные рекомендации помогают избежать конструкции if/else в большинстве случаев, однако есть ситуации, когда избежать этого нельзя.

**Ветвления**

В состав современных процессоров входит модуль предсказания ветвлений – branch predictor. Всякий раз, когда код имеет ветвь (например, if-структуру) или если мы находимся внутри цикла, микропроцессор не знает заранее, какая из двух ветвей нужна для подачи в pipeline. Branch predictor нужен для того, чтобы предсказать по какой ветке пойдет if – останемся ли мы внутри тела цикла или выйдем из него.

Сам процессор работает быстрее, чем оперативная память, из которой он берет данные. Поэтому желательно обращаться к памяти таким образом, чтобы данные заведомо находились либо в регистрах процессора, либо в кэш памяти. В таком случае, процессор может эти данные подтянуть, если будет знать, куда пойдут вычисления.

Вычисления развиваются линейно, поэтому предсказать, что понадобится через некоторое количество тактов процессора достаточно легко, если в этом коде нет ветвлений.

Например, конструкция ветвления с оператором if – самый плохой случай, т.к. оператор if предусматривает различные длинные последовательности вычислительных команд.

Пример конструкции ветвления с оператором  if:

**if** (num < 10) { *// Если введенное число меньше 10.*

cout << "Это число меньше 10." << endl;

} **else** { *// иначе*

cout << "Это число больше либо равно 10." << endl;

}

Усовершенствуем программу следующим образом:

**if** (num < 10) { *// Если введенное число меньше 10.*

cout << "Это число меньше 10." << endl;

} **else** **if** (num == 10) {

cout << "Это число равно 10." << endl;

} **else** { *// иначе*

cout << "Это число больше 10." << endl;

}

`Рассмотрим короткие ветвления, которые в большинстве случаев имеют преимущества над длинными. Их разумно выполнять в виде конструкции – с оператором “?:” ( x = a>b ? y : z; ).

Пример:

if (a > 0)

{x = 1, y = 2, z++ }

else {q = 3 z = 4 y = 5}

Можно переписать с помощью одной тернарной операции.

Для начала нужно найти общие присваемые части в обоих случаях – это z и y. В z выполняются разные операции: z++ - инкрементируется; z = 4 записывается. А y в обоих случаях просто записывается, что довольно удобно.

y = a > 0 ? { x = 1, z++, 2} – непрерывная тернарная операция

y = a > 0 ? ( x = 1, z++, 2) : ( q = 3, z = 4, 5)

В данном методе исключается предсказатель ветвления – просто выполняется логическое выражение. Данный метод выигрывает в скорости выполнения.

y = a > 0 ? 2:5;

x = a > 0 ? 1:x;

z = a > 0 ? z++:4;

q = a > 0 ? q:3

Другим методом оптимизации ветвлений является устранение ветвления, а именно замена на выбор из таблицы. Вместо условного выражения, в котором оба случая являются константами, используется справочная таблица. Если есть оператор, подобный следующему, где c и d представляют константные выражения, а b представляет логическое выражение:

a = b ? c : d

Это эквивалентно следующему:

if (b) a = c;

else a = d;

Заменим следующим эквивалентным и более быстрым кодом:

static const type lookup\_table[] = { d, c };

a = lookup\_table[b];

Условное выражение компилируется в ветвь. Если такая ветвь не очень хорошо предсказана, это займет больше времени, чем таблица соответствия.

a = b1 ? c : b2 ? d : b3 ? e : f;

Это эквивалентно следующему:

if (b1) a = c;

else if (b2) a = d;

else if (b3) a = e;

else a = f;

Заменим следующим эквивалентным кодом:

static constant type lookup\_table [] = { f, e, d, d, c, c, c, c };

a = lookup\_table [b1 \* 4 + b2 \* 2 + b3];

Оператор switch. switch, всегда можно заменить на константный массив.  
  
Под последовательными значениями представляется:

switch (a) {

case 0:

case 1:

case 2:

…

}

Безусловно switch работает быстрее многоуровневых if else, но все-равно это условие. Для повышения производительности всегда можно воспользоваться константным массивом.  
Пример:

int a, b = 0;

/\*switch (a) {

case 0: b\* = 10;

case 2: b\* = 20;

case 3: b\* = 30;

…

}\*/

const int array[3] = {10,0,20,30};

b \*= array[a];

Самый основной (неоптимизированный) способ составления оператора switch - это трактовать его как цепочку if … else if … операторов. Обычный способ, которым компиляторы оптимизируют переключатель, - это преобразовать его в таблицу переходов, которая может выглядеть примерно так:

if (condition1) goto label1;

if (condition2) goto label2;

if (condition3) goto label3;

else goto default;

label1:

<<<code from first `case statement`>>>

goto end;

label2:

<<<code from first `case statement`>>>

goto end;

label3:

<<<code from first `case statement`>>>

goto end;

default:

<<<code from `default` case>>>

goto end;

end:

Одна из причин, по которой этот метод работает быстрее, заключается в том, что код внутри условных выражений меньше (поэтому при неправильном прогнозировании условного кэша будет меньше штраф). Кроме того, «провальный» случай становится более тривиальным для реализации (компилятор пропускает goto end оператор).

Компиляторы могут дополнительно оптимизировать таблицу переходов путем создания массива указателей (на места, отмеченные метками) и использовать значение, которое вы включаете, в качестве индекса в этом массиве. Это исключило бы почти все условные выражения из кода (за исключением того, что было необходимо для проверки того, соответствует ли значение, которое вы включаете, одному из ваших случаев или нет).

Важно отметить, что вложенные таблицы переходов трудно генерировать, и некоторые компиляторы отказываются даже пытаться их создать. По этой причине стоит избегать вложения switch внутрь другого, switch если важен максимально оптимизированный код.

**Оптимизация массивов**

Массив – это структура данных, представленная в виде ячеек определенного типа, объединенных одним именем. Массивы используются, когда необходимо обработать большое количество данных. Его имя является постоянным указателем, который указывает на первый элемент массива. Вследствие чего можно использовать указатели и их арифметику.

Обращение к элементам массива выглядеть, как показано в примере 1.

Пример 1.

for (int i=0; i<n; i++)

array [i] = n\_some\_value;

Вместо приведенного выше кода эффективнее сделать, как показано в примере 2.

Пример 2.

int\* arrn=array +n;

for (int \* ptr\_int=array; ptr\_int<arrn; ptr\_int ++)

\* ptr\_int= n\_some\_value;

Причина этого в операциях с указателями. В примере 2 есть указатель на тип int, который берет адрес из имени массива array и увеличивает его для каждого элемента.

В примере 3 рассмотрена очевидная обработка двумерного массива. ( В данном случает его обнуление).

Пример 3

for (int i=0; i<n\_rows; i++) {

for (int j=0; j<n\_cols; j++) {

array[i][j]=0;

}

}

Использование указателей сделает код из примера 3 более эффективным.

Пример 4

int \*p = &array[0][0];  
int \*end = &array[0][0] + n\_rows \*n\_cols;  
for (p = [\*array](https://vk.com/array); p < end; p++) {  
\*p = 0;  
}

Используя оптимизацию из примера 4, можно ускорить обработку массива приблизительно в 10 раз. Вместо того, чтобы вычислять позицию каждого элемента с помощью индексов, указатель перемещался по массиву, увеличивая на одну целую позицию после каждой итерации до конца – адреса, который был вычислен один раз, заранее. Конечно, хороший оптимизирующий компилятор упростил бы код примера 4.

В примере 5 требуется посчитать сумму элементов двумерного массива

Пример 5.

int a[n][m];

int sum =0;

for (int i=0; i<n;i++){

for (int j=0; j<m;j++ )

sum+=a[i][j];

}

Время выполнения фрагмента кода из примера 5 может сильно увеличить

при достаточно больших n и m, потому что на каждой итерации цикла при

определении адреса [i][j] – го элемента компилятор использует много

вычислений.

Зная о том, что элементы массива расположены в памяти последовательно,

можно оптимизировать пример 5, введя вспомогательный указатель.

Пример 6.

int \*p= &a[0][0];

float size=sizeof(a)/sizeof(int);

for (int i=0; i<size;i++){

sum+=\*p;

p++;

}

Так же можно использовать эквивалентное объявление указателя \*p= a[0].

Пример 7.

int \*p; //указатель на массив элементов

x=\*(p++);

у=\*(++p);

В примере 7 первое присваивание значительно эффективней, чем второе. В первом случае будет осуществляться разыменование указателя и его инкремента параллельно, а во втором – последовательно.

Еще один совет по оптимизации работы с двумерными массивами.

Количество столбцов двумерного массива желательно должно быть равно степени двойки при небольших типах данных. Это ускорит работу с массивом, поскольку это выравнивает указатели на первые элементы каждой строки, что ускоряет доступ к элементам.

Пример 8.

double array [7][8]; // 8 =23

**Оптимизация switch**

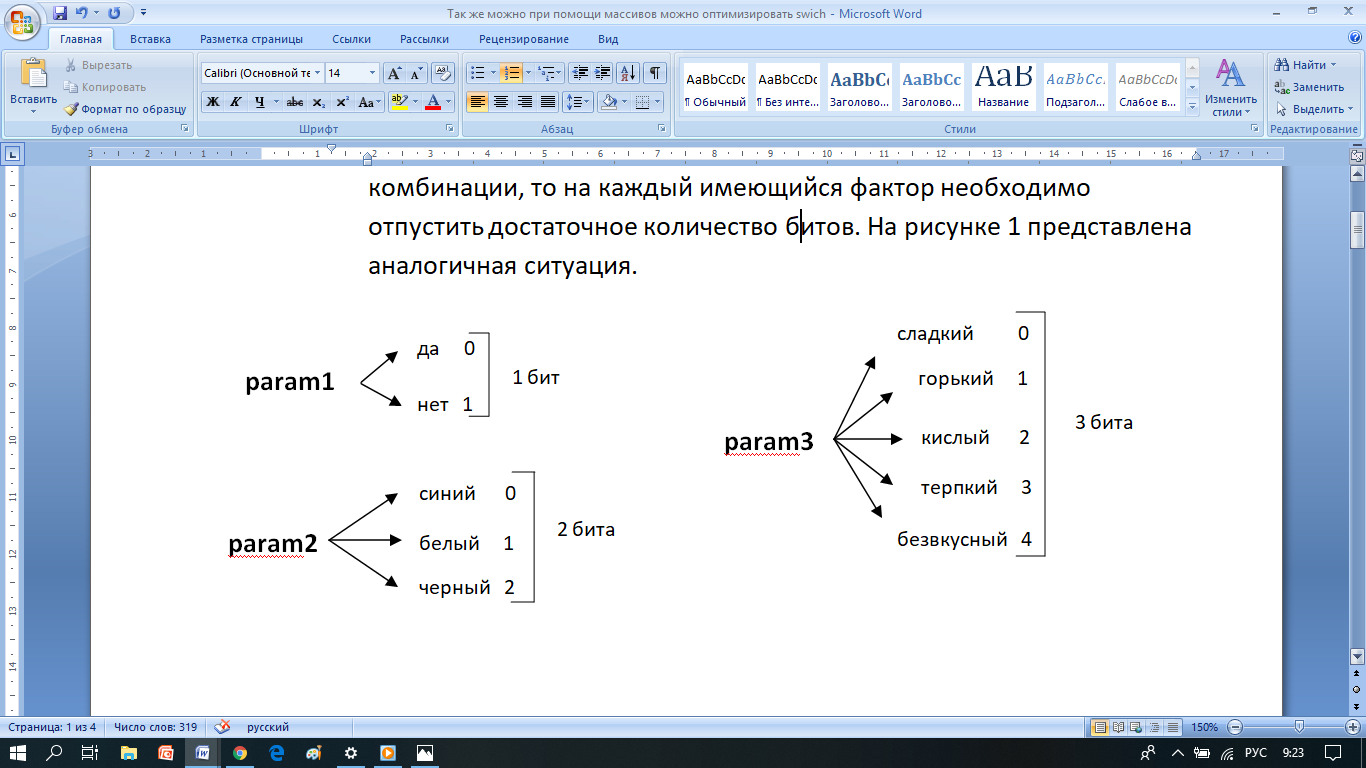
Задача: имеется много разных параметров (**param**), от которых зависит решение задачи. Все параметры независимы друг от друга. Каждый из них принимает конечное количество вариантов.

Поставленную задачу можно решить несколькими способами:

I способ. *Кодирование всех возможных вариантов.*

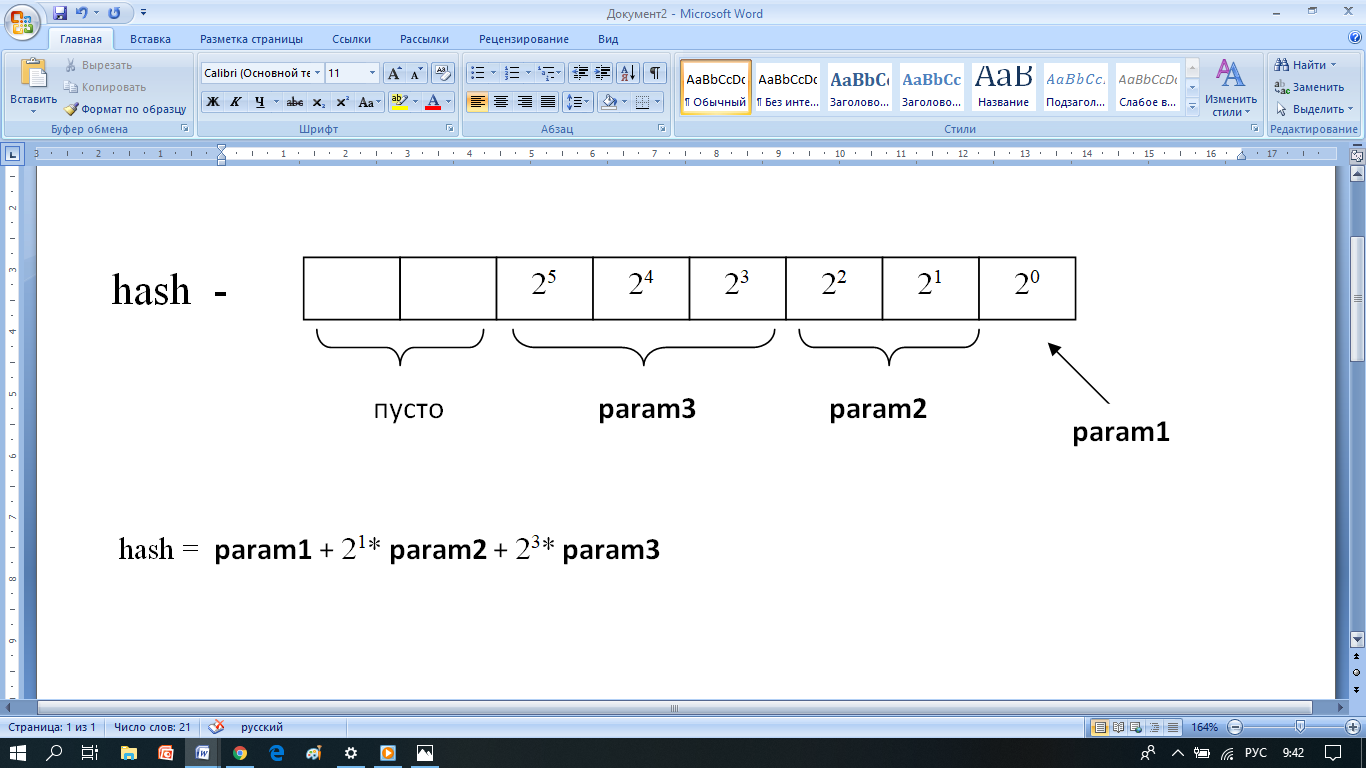
Этот способ применяется только тогда, когда возможно закодировать все возможные варианты.

1. Каждый имеющийся параметр необходимо закодировать достаточным (но не более того) количеством битов. На рисунке 1 представлен пример подобной ситуации.



*Рисунок 1. Кодирование факторов*

1. Из этих коротких целых чисел, каждое из которых содержит маленькое количество битов, при помощи сдвигов собирается одно число (hash), которое в итоге попадает на выражение switch.



*Рисунок 2. Формирование hash*

II способ. *Кодирование только возможных сочетаний.*

Этот способ применятся только тогда, когда нет возможности закодировать все варианты, так как некоторые из них невозможны. Например, не все варианты param1 можно сопоставить со всеми вариантами параметра param2 и т.д. В этом случае hash можно образуется следующим образом:

1. логическое выражение
2. a = a0 + (сложное логическое выражение 1)\* a1

b = b0 + (сложное логическое выражение 2)\* b1

c = c0 + (сложное логическое выражение 3)\* c1

Например, если положить a0 = b0 = c0 = 0, а a1 = b1 = c1 = 1, то

hash = a+2\*b+4\*c

с) закодировать возможные сочетания при помощи индексного массива (index arrays/ Look Up Table).

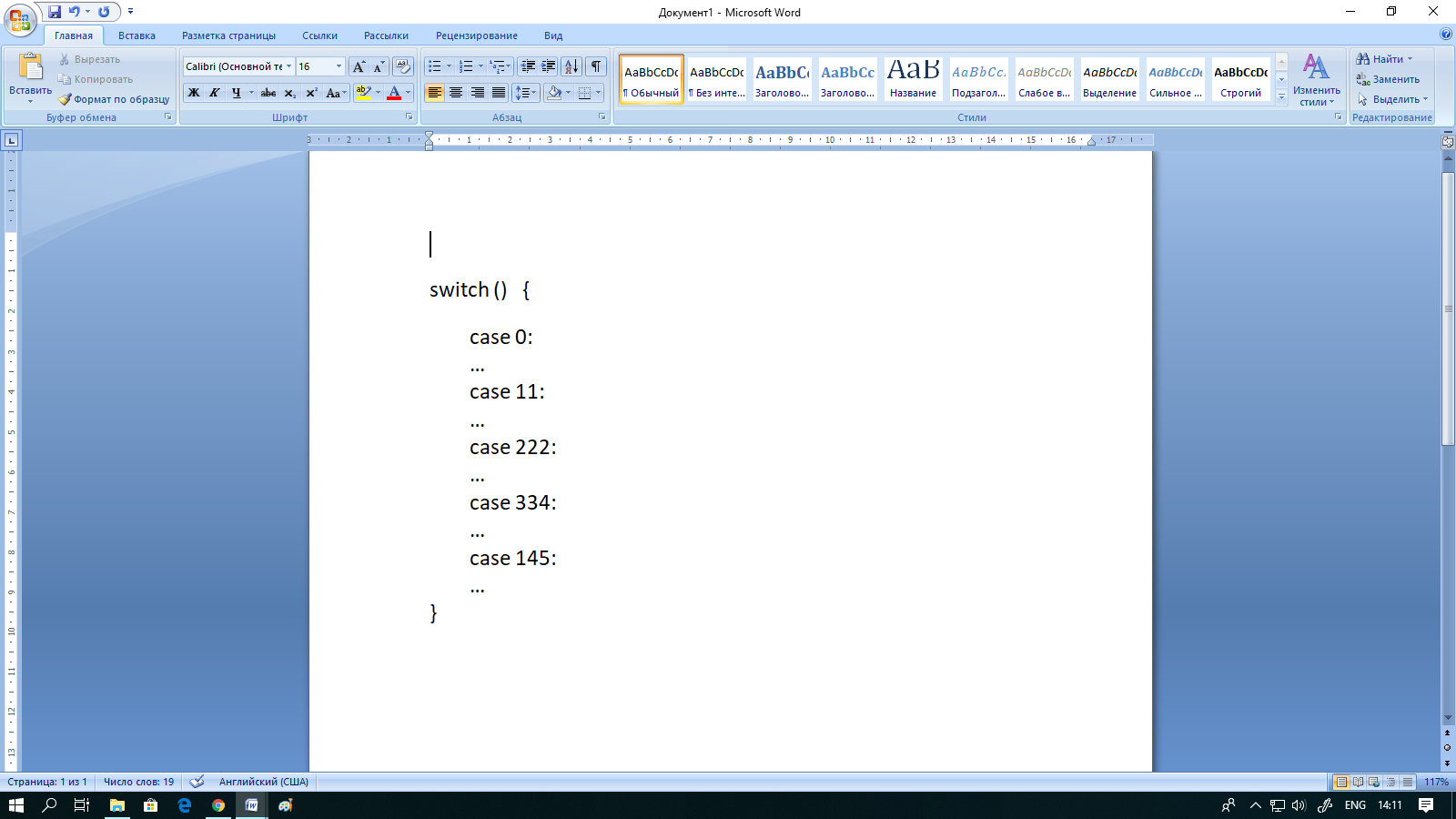
1. Создадим вспомогательный массив. Исходные ситуации описываются числами, которые идут на вход switch. На рисунке 3 представлена визуализация индексного массива.



*Рисунок.3. Индексный массив*

Это необходимо для того, чтобы непрерывно идущий числовой ряд (элемент 0, 1, …,4) преобразовать в необходимые для switch значения, которые могут идти в различном порядке.

1. Выполняем промежуточную перекодировку (рисунок 4).



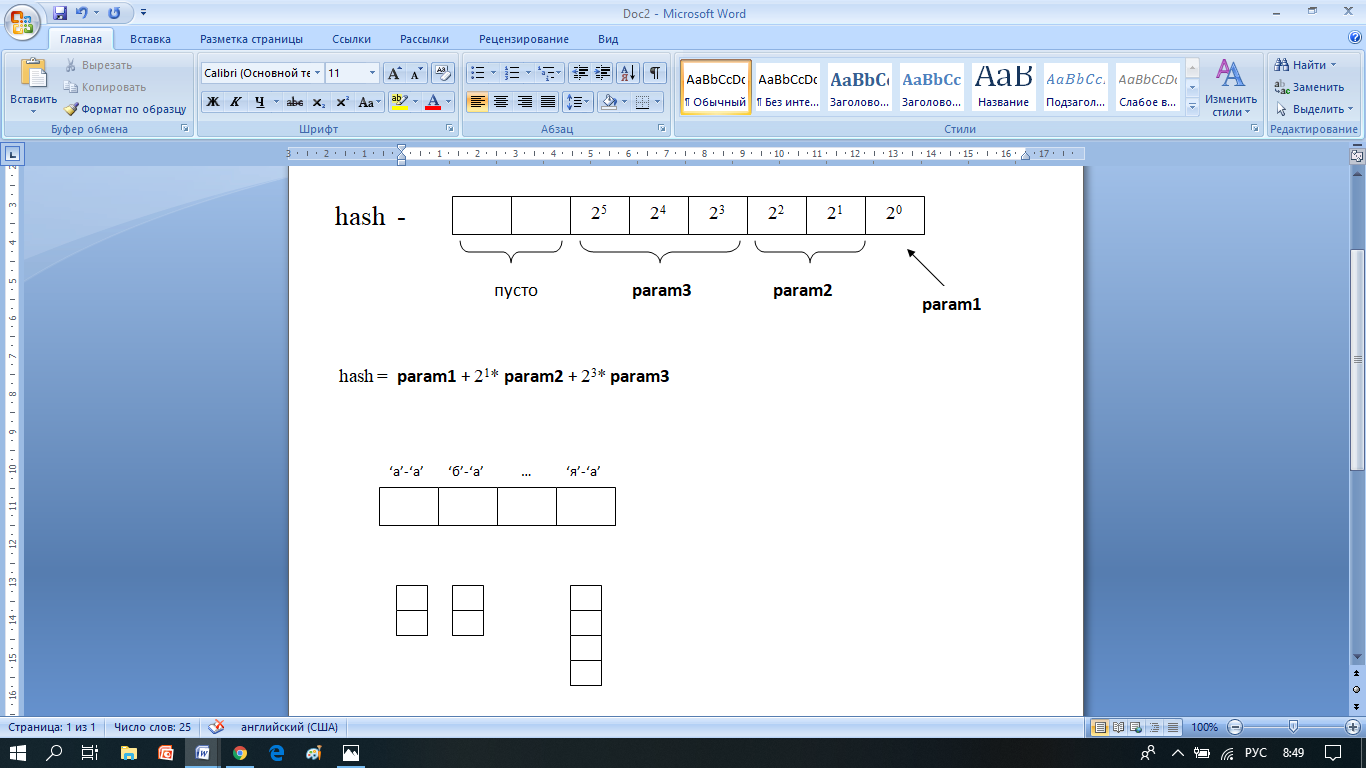
*Рисунок 4. Использование индексного массива*

Рассмотрим, как можно применить данный метод на практике.

*Задача*. Есть определенное правило, по которому осуществляется транслитерация, т.е. на вход поступает русский текст, известно правило, по которому всякой русской букве или буквосочетанию можно поставить в соответствие латинскую букву или буквосочетание. Необходимо преобразовать входной тест в соответствии с заданным правилом. (Например, ‘с’ ‘s’, ‘ш’ ‘sh’).

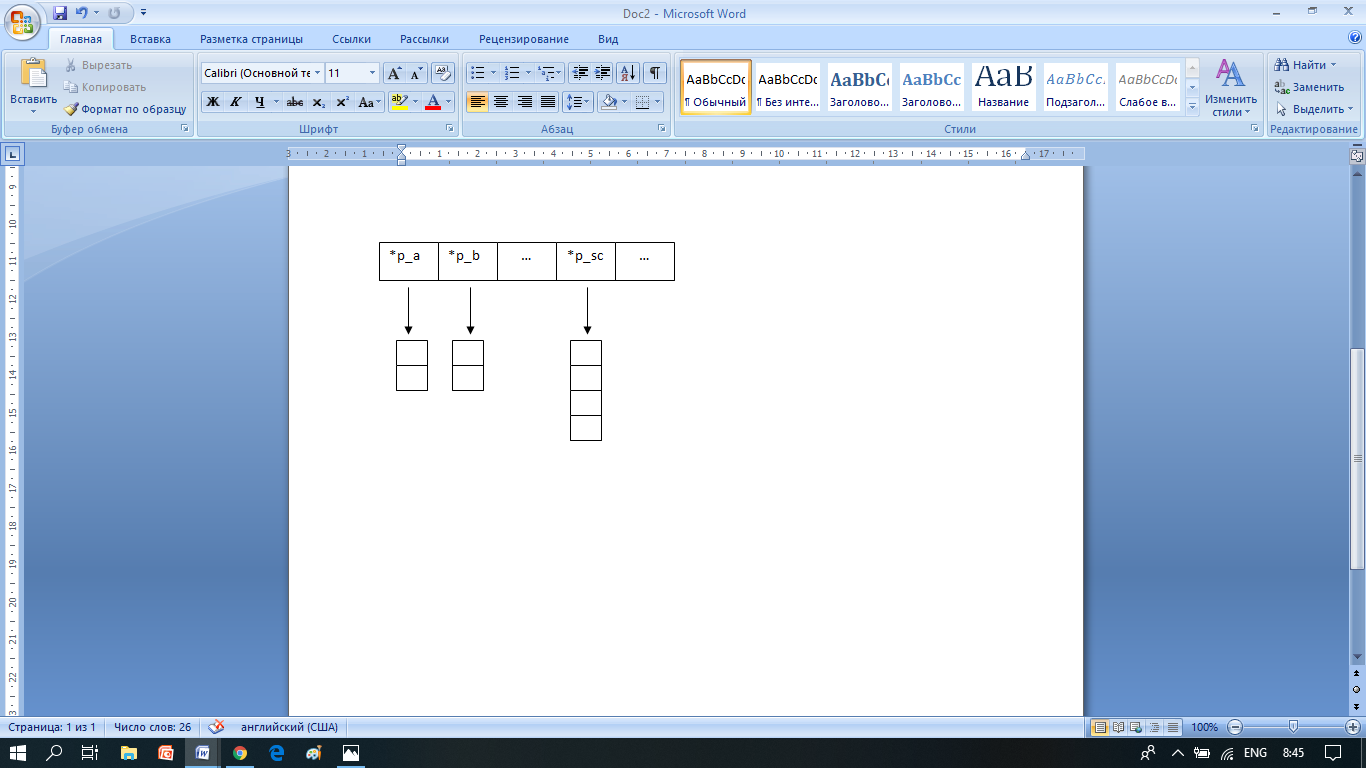
Нужно максимально оптимизировать транслитерацию.

1. Формируем вспомогательный массив таким образом, чтобы индексы массива получались посредством вычитания из кода буквы кода буквы ‘а’.



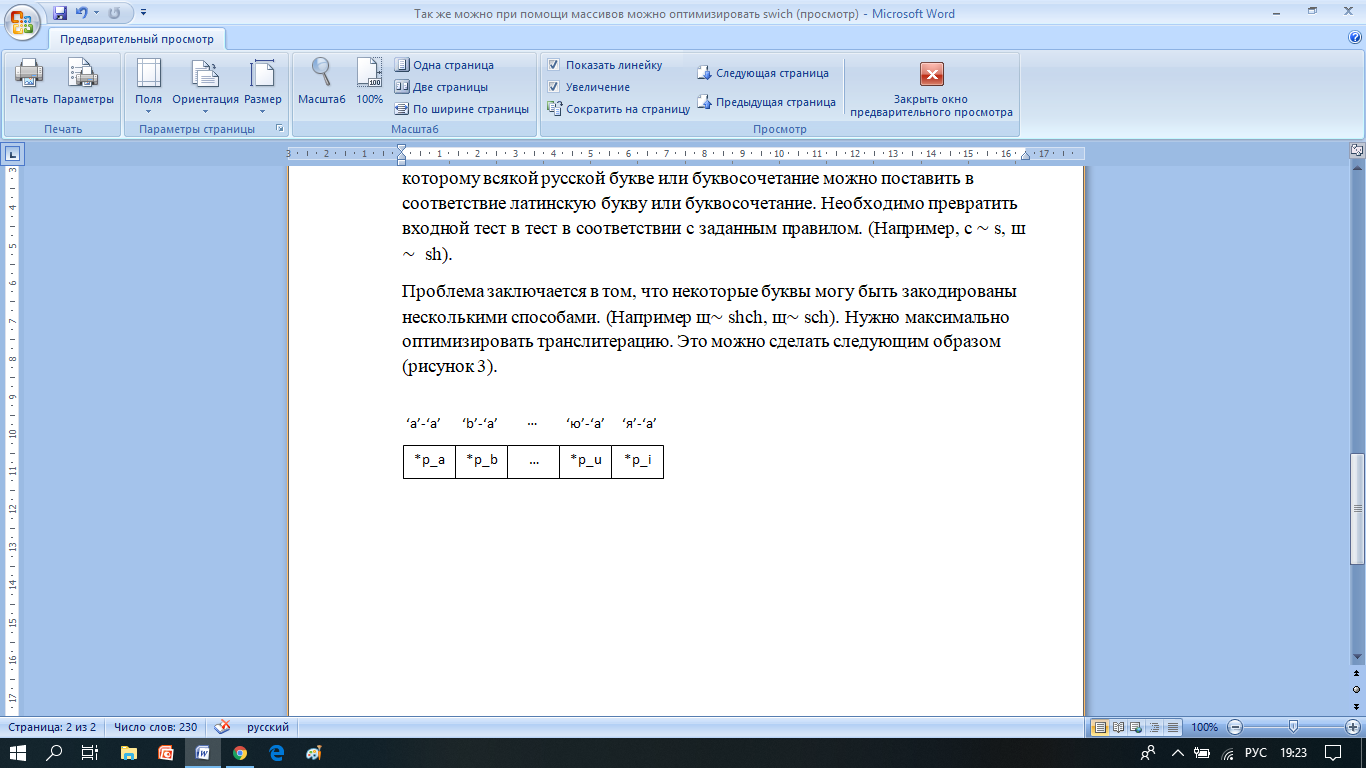
*Рисунок 5. Индексы массива*

1. В каждую ячейку массива помещаем указатель на массив, где хранится транслитерация отдельно взятой буквы.
2. Массивы с транслитерацией буквы могут быть разной длины, потому что некоторые буквы транслитерируются несколькими символами. Например, ‘щ’ ‘sch’ (рисунок 6).



*Рисунок 6. Указатели на массивы*

1. Результат изображен на рисунке 7, а в примере 1 представлен фрагмент кода.



*Рисунок 7. Использование индексного массива для транслитерации*

Пример 1.

switch (character){  
 case 'а'-'а':  
 \*p\_a;  
 break;  
 ...  
 case 'ж'-'а':  
 \*p\_zh;  
 break;  
 …   
}

# **Устройство кэш памяти**

* **Кэш память** – сверхбыстрая память, расположенная ближе всех к процессору. Обычно она быстрее оперативной памяти примерно в 2 или более раз. Хранит в себе копии часто используемых данных из обычной памяти.
* Кэш память обычно представляет собой трехуровневую систему:

Первый уровень(L1) – самая быстрая память, но самая маленькая. Нет задержек, бывает двух типов: пространство инструкций и пространство данных. На каждое ядро выделено по каждому из этих типов.

Второй уровень(L2) – немного медленней L1, но объем ее при

этом в десятки раз больше чем у L1. Хранит в себе менее часто используемые данные, по сравнению с L1, нет разделения на инструкции и данные. Для каждого ядра процессора устанавливается свой кэш L2, то есть у одного ядра нет доступа к кэшу другого.

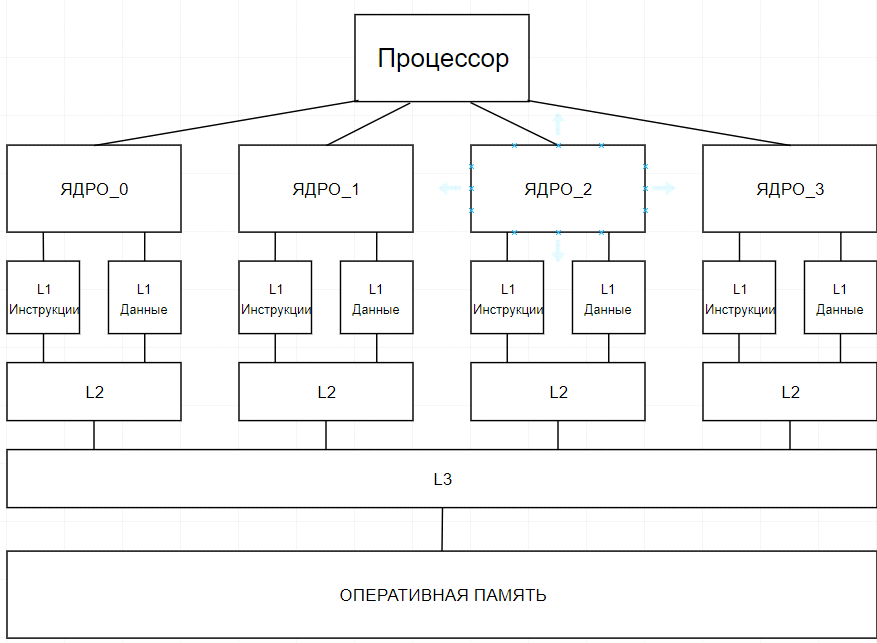


Рис 1. Архитектура кэш памяти процессора

Третий уровень(L3) – медленнее, чем L2, но по объему является самой большой. Размер может достигать нескольких мегабайт. Является общей для всех ядер процессора.

* Контроллер кэш-памяти.

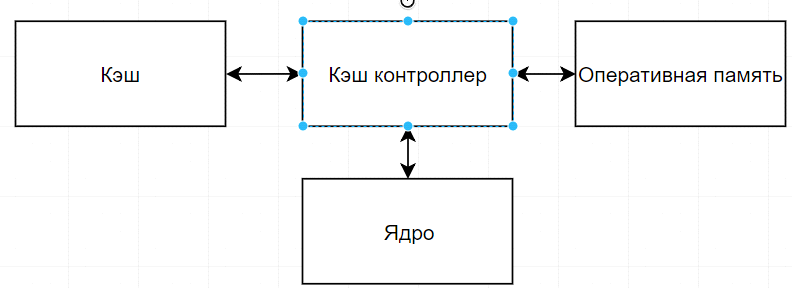


Рис 2. Взаимодействие контроллера с процессором и памятью

Это устройство, передающее данные между кэш памятью, оперативной памятью и процессором. Когда ядро обращается за данными, контроллер проверяет их наличие в кэш памяти. Если данные на месте, то происходит немедленная передача данных ядру, в противном случае (*промах кэша-cache miss*) придется ожидать поступления данных из оперативной памяти. Задача контроллера – сделать так, чтобы промахов кэша было как можно меньше.

Рис2. Взаимодействие контроллера с процессором и памятью

4.**Организация кэш памяти**

Кэш-память — это набор кэш-строк. Каждая кэш-строка хранит блок данных определенного размера и дополнительную информацию (например, тэг). Под размером кэш-строки понимают обычно размер блока данных, который в ней хранится.

Для задания адреса каждого байта памяти требуется x-байтный адрес. Предположим, что наш кэш работает на уровне отдельных байтов, то есть может сохранять в качестве элемента байт оперативной памяти. Тогда каждая единица кэша должна сохранять байт данных оперативной памяти и его x-байтный адрес в оперативной памяти. Получается строка, которая называется кэш­строкой. Адрес сохраняемого байта – тэг(tag). При чтении данных из кэша процессор формирует адрес, который сравнивается кэш-контроллером с тегом кэш­строки. В случае совпадения кэш-контроллер выдает требуемый байт данных, если же совпадения адреса с тегом нет (кэш­промах) — производится обращение к оперативной памяти, что не есть хорошо, так как ядро. Примечание: размер адреса – х, зависит от разрядности системы.

* Подробнее о тэгах, их оптимизации.

Тэги очень дорого обходятся для системы по следующим причинам:

* логика сравнения тегов большая (операция XOR занимает лишний бит)
* логика сравнения тегов медленная
* слишком большие затраты памяти на хранение тэгов

Например, в полностью ассоциативном отображении RAM на кэш запись тега в кэш обходится в 30 битов, а запись данных к этому тэгу – 32, то есть 48% памяти кэша расходуется на хранение тэгов!

4.1. Уменьшение затрат на тэг: Больше данных на один тэг

Сделать кэш-линии больше: увеличить каждую строку в полностью ассоциативном кэше.

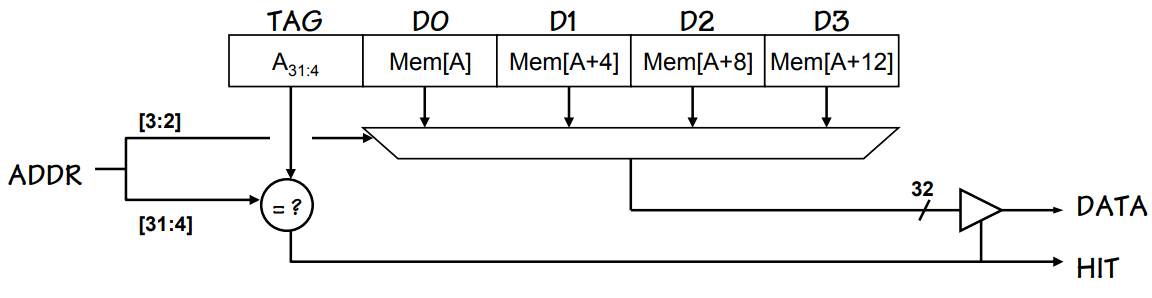


Рис 4. Увеличение кэш-линии

* Блоки из двоичных слов
* всегда читает/записывает двоичный слово-блок из/в память
* использует пространственную локализацию: соседние слова в блоке, вероятно, доступны
* стоимость: присутствуют некоторые наборы недоступных слов
* большой профит, если путь к памяти широк или последовательный доступ быстрый

В итоге получается, что на хранение тэгов необходимо 32 - 2 = 30 битов, а на хранение блока данных 4\*32 бита. Отсюда следует, что всего 18% от всей кэш памяти использовано на хранение тегов.

4.2. Соотношение промахов кэша(cache miss) к размеру блоков

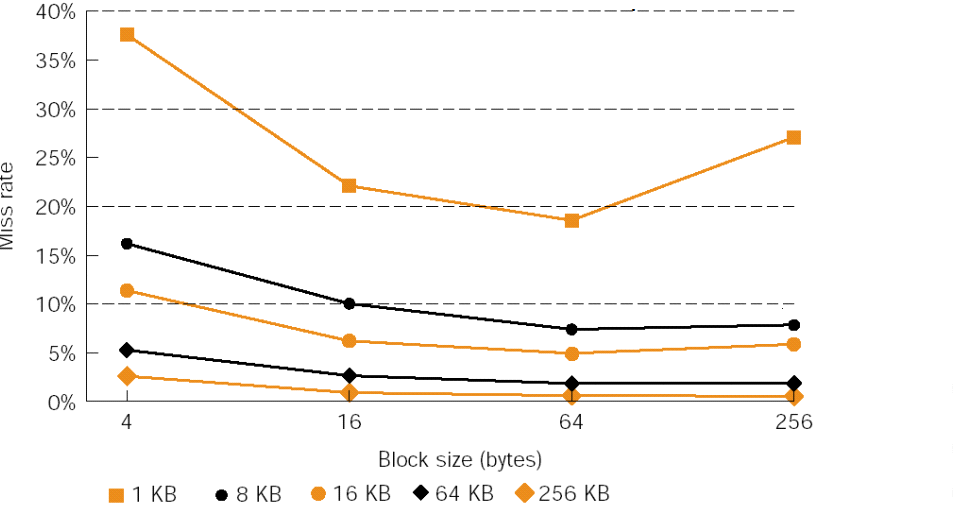


Рис 5. Соотношение промахов к размеру блоков

Если блоки данных сделать очень большими, то количество промахов

кэша в конечном итоге сильно увеличится и потому следует использовать

оптимальный размер блоков.

## 5. **Три различные стратегии по заполнению кэша.**

1) LRU (Least-recently used): заменяет объект, к которому дольше всего не обращались и предпочитает самые часто используемые данные. Лучшая производительность / высокая стоимость;

2) FIFO/LRR (first-in, first-out/least-recently replaced): заменяет самый старый элемент в кэше, предпочитает недавно загруженные элементы старым. Низкая производительность / экономичность;

3) Random: заменяет объекты случайным образом, нет фаворитов – все равны, использует псевдослучайный генератор, чтобы получить воспроизводимое поведение. Средняя производительность / самая низкая стоимость, избегает патологических последовательностей, но производительность может варьироваться.

## 6. **Отображение RAM на кеш-память**

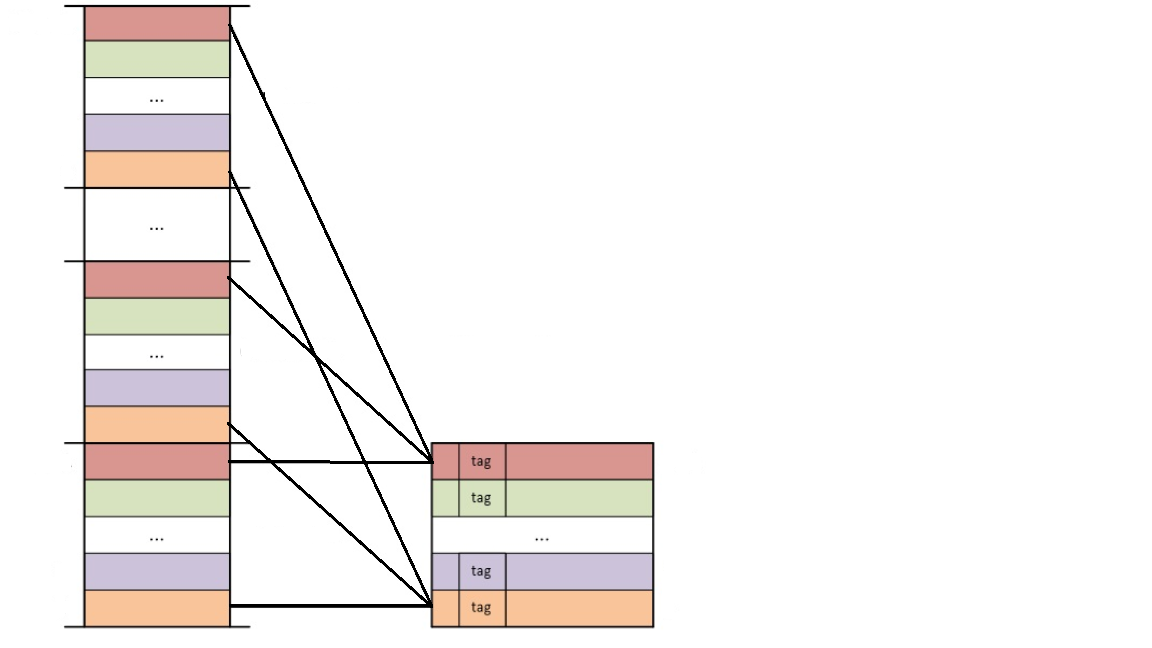
1) Direct-Mapped Cache (Прямое отображение): RAM делится на сегменты, размер каждого сегмента равен размеру кэша, а каждый сегмент в свою очередь делится на блоки, размер каждого блока равен размеру кэш строки. Блоки RAM из разных сегментов, но с одинаковыми номерами в этих сегментах, всегда будут отображаться на одну и ту же кэш строку. 

Рис 6. Устройство прямого отображения

Использовать стратегию для заполнения нет нужды, так как каждый блок ссылается на определенную кэш строку. Для хранения тэгов/данных используется SRAM.

2)Fully Associative Cache (Полностью ассоциативное отображение): RAM делится на блоки, размер которых равен размеру кэш строки. Каждый блок имеет своё сравнивающее устройство и может отображаться в любую кэш строку. Устройство сравнивает адрес с тегом кэш строки, в случае успеха возвращает данные процессору.

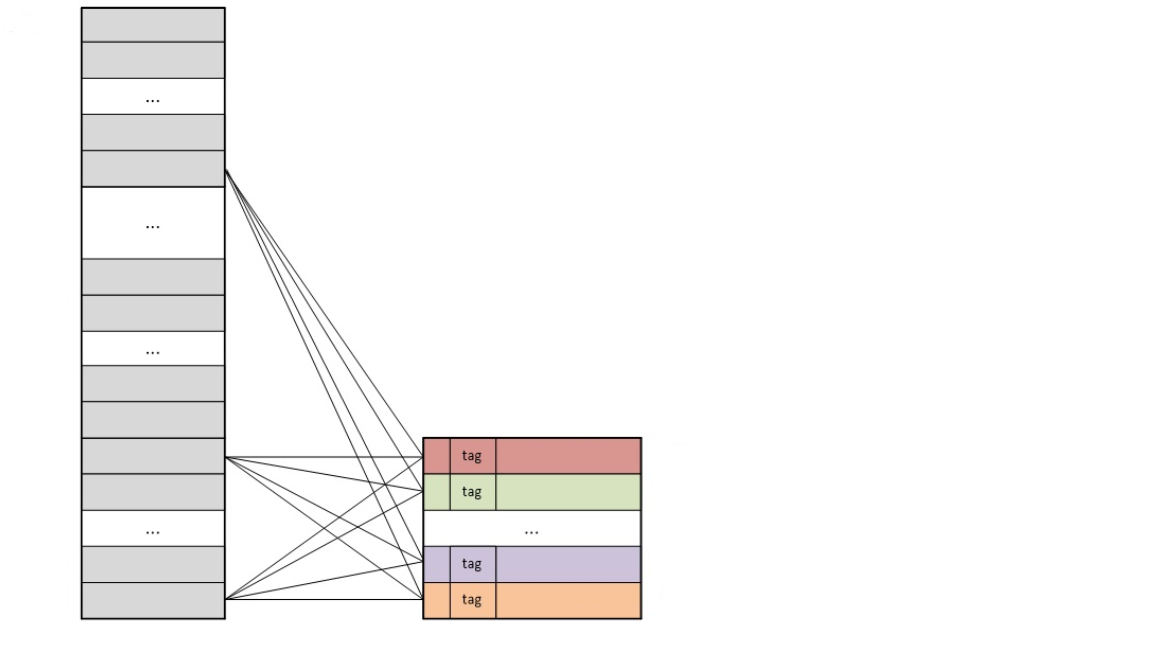


Рис.7 Устройство полностью ассоциативного отображения

Используются стратегии заполнения памяти для поиска кэш строки при добавлении новых данных. Для хранения тэгов/данных используются регистры.

3) N-Way Set-Associative Cache (Наборно-ассоциативное отображение): совмещает в себе два предыдущих способа отображения. RAM делится также, как и в прямом отображении, а сам кэш состоит из k кэшей, использующих прямое отображение. Кэш строки, имеющие одинаковые номера во всех каналах, образуют набор.

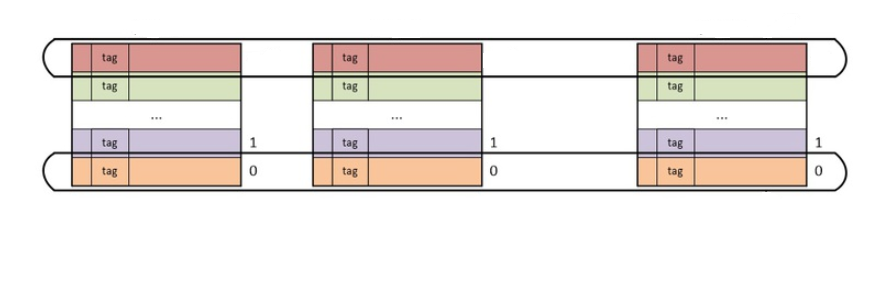


Рис.8 Набор в наборно-ассоциативном отображении

Набор представляет собой кэш с полностью ассоциативным отображением. Блоки RAM из разных сегментов, но с одинаковыми номерами в этих сегментах, всегда будут отображаться на один и тот же набор кэша.

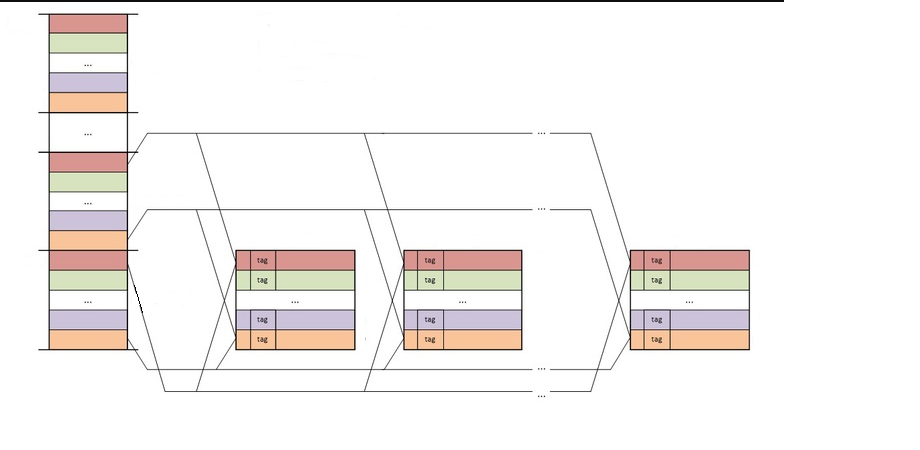


Рис.9 Устройство наборно-ассоциативного отображения

6.1 Соотношение промахов к ассоциативности

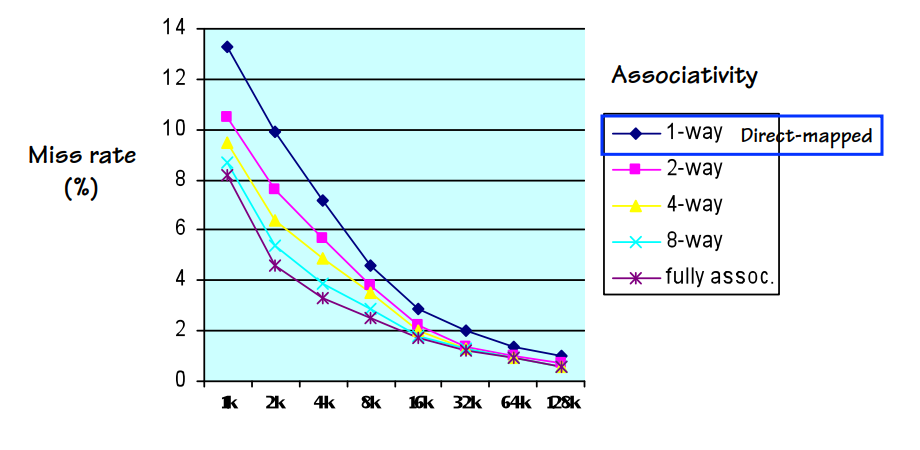


Рис.10 Соотношение промахов к ассоциативности

Видим, что более ассоциативное отображение способствует меньшее количество промахов.

**Выравнивание данных (data alignment)**

Выравнивание ускоряет доступ к памяти за счет генерации кода, в котором на чтение и запись ячейки памяти требуется по одной инструкции. Без выравнивания мы можем столкнуться с ситуацией, когда процессору придется использовать две и более инструкции для доступа к данным, расположенным между адресами, кратными размеру машинного слова. char — особый случай, они занимают ровно одно машинное слово и всегда требуют одинакового количества инструкций для доступа. Поэтому для них нет предпочтительного выравнивания. Сhar может начинаться с любого адреса, однако двухбайтовый short должен начинаться только с четного адреса, четырехбайтный int или float — с адреса, кратного 4, восьмибайтные long или double — с адреса, кратного 8. Наличие или отсутствие знака значения не имеет. Указатели — 32-битные (4 байта) или 64-битные (8 байт) — также выравниваются.



При чтении данных с адреса 0 процессору требуется две операции.

А при считывании данных с адреса 1, так как адрес не кратен размеру считываемых данных, процессору приходится выполнять лишнюю работу, считывая при этом 6 байт вместо 4. Такой адрес называется не выровненным. При доступе к данным по не выровненным адресам процессору приходится делать лишние обращения к памяти, что в итоге приводит к снижению скорости работы программы.

Методы:

1. Использование \_\_declspec( align( # ) )

\_\_declspec( align( # ) ) declarator

# — значение выравнивания. Допустимые записи — целые степени двойки. declarator — данные, которые объявляются как выровненные.

\_\_declspec ( align(#) ) можно использовать при определении struct, union или class либо при объявлении переменной. Однако, \_\_declspec(align(#)) влияет только на выравнивание структуры, а не каждого поля в ней. Если необходимо выровнять каждый член структуры, нужно указать выравнивание на уровне элементов или путем введения дополнительных переменных.

Например, можно запросить 64-байтное выравнивание для массива следующим образом:

\_\_declspec( align( 64 ) ) double a[N];

1. Заполнение

char \*p;

char c;

int x;

Рассмотрим, как располагаются в памяти данные переменные. Память для p начинается с адреса, кратного 4. Выравнивания указателя — самое строгое. Следом за ним идет c. Но четырехбайтный x требует заполнения пустыми байтами. Происходит примерно то же самое, как если бы была добавлена еще одна переменная:

char \*p /\* 4 or 8 байт \*/

char c /\* 1 байт \*/

char pad[3] /\* 3 байт \*/

int x /\* 4 байт \*/

Массив символов pad[3] в данном случае указывает на то, что пространство заполняется тремя пустыми байтами.

Если тип x будет short, который занимает 2 байта, данные будут располагаться так:

char \*p /\* 4 or 8 байт \*/

char c /\* 1 байт \*/

char pad[1] /\* 1 байт \*/

short x /\* 2 байт \*/

С другой стороны, на 64-битной машине эти данные расположатся в памяти следующим образом:

char \*p /\* 8 байт \*/

char c /\* 1 байт \*/

char pad[7] /\* 7 байт \*/

long x /\* 8 байт \*/

Случай, если переменная с меньшим размером будет объявлена в начале:

char c;

char \*p;

int x;

Чтобы эти переменные занимали меньше памяти, можно поменять местами x и c:

char \*p /\* 8 байт \*/

long x /\* 8 байт \*/

char c /\* 1 байт \*/

Путем реорганизации скалярных величин можно сэкономить несколько байт памяти.

**Все данные размером машинного слова**

В современных процессорах, в отличии от компиляторов, выравнивание данных присутствует всегда, происходит оно по размеру машинного слова. Машинное слово (от англ. «word» - слово) – это количество данных, которые процессор может обработать за одну операцию. Слово – это некоторое количество байт, если точнее, то байт. Его размер указывается в разрядности процессора, которую мы часто слышим, к примеру, Intel Pentium – 32 разрядный процессор, то имеют в виду, что длина машинного слова равняется 32 бита или 4 байта.

Процессор может адресовать данные исключительно с квантом, равным длине его регистра. Размер процессорных регистров общего назначения равен размеру машинного слова этого процессора. То есть, если у вас 64 разрядный процессор, то регистры будут иметь длину 32 бита, либо 64. Например, если у процессора имеется 32 разрядный регистр для хранения Integer’a, то это значит, что все целые числа он хранит посредством погружения в эти 32битные регистры, то есть для хранения типа «двойного длинного слова» – long long (64 бита) будут использоваться два подряд идущих регистра. Но при этом, для хранения адресов может быть 64 битный регистры, это значит, что все адреса могут начинаться только с адреса кратным 8 байт (64 бита). Таким образом разрядность в отношении целых чисел может быть не равна разрядности в отношении адресов, либо чисел с плавающей точкой (float), потому что с ними работают разные регистры.

В итоге, если использовать для данных только регистры размером машинного слова, то вы работаете быстро, всегда ли это оправдано? Не всегда. Как правило, в большинстве ситуаций, стоит гоняться за выигрышем скорости, но при это вы проигрываете в памяти, и избежать этого никак не получится.

**Борьба с глобальными объектами**

Часто часть данных, которая нужна самым разным функциям в вашей программе оформляют в виде глобальных данных.   
  
Проблема заключается в том, что глобальный объект может использоваться одной и той же функцией (или разными) на разных процессорах.   
  
Рассмотрим 2 случая, чтобы увидеть проблему глубже:   
  
**1. Глобальный объект является константой**. Вы не можете его поменять и на первый взгляд ничего плохого нет.

**Первая проблема**. Объект может требоваться разным процессорам, в кэши которых копируется одновременно. Если он будет копироваться в разные кэши разных процессоров, то это приведет к многочисленным копиям глобального объекта. В нем много информации, включаю ту, которая не нужна в данный момент.

**Вторая проблема**. Вам требуется улучшение исходного текста. К примеру, сделать так, чтобы каждый [инстанс](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%81) получал свой вариант этого объекта. Тогда логично не делать глобальный объект, а передавать через указать каждую конкретный экземпляр.

**2. Глобальные объект не является константой.** Проблем возникает больше, и они имеет более значительные последствия.

**Первая проблема**. К примеру, одна функция стала менять в вашем объекте какое-то поле, и оно не защищено от синхронных изменений, если вы того специально не сделали. Тогда может возникнуть ситуация, когда поменяли одну половину объекта, а вторая осталось прежней. Это называется нарушением атомарности вычислений.

**Вторая проблема** вытекает из первой, если вы будете защищать объект синхронизации, но он глобальный. В этом случае [мьютекс](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81) блокирует доступ, а все остальные функции будут ждать. Такую ситуации можно сравнить с большим грузовиком, который встал посередине эстакады. Избежать проблемы можно, разбив глобальный объект и иметь его в нескольких копиях, передавая функции указатель на одну из них.

Как итог, глобальных объектов стоит избежать, даже если мы, с помощью методов выше, сделаем его работу корректной, он не будет эффективным.

Попытайтесь разбить на несколько частей и передавать указателями, не используя глобальных объектов. В ряде случаев, это невозможно, так как **универсального совета не существует**.

**Ускорение вызова функций**

Каждая программа состоит из кода и вызова функций. При вызове функций возникают накладные расходы, такие как: передача параметров и управление им, создание и удаление внутренних переменных, возвращение управления и результата. Чтобы минимизировать данные расходы используют такие метода как макросы и inline-подстановки. Они похожи по производимому эффекту, но различаются в плане реализации.

Рассмотрим эти два метода подробно.

1. **Макросы**

Макрос – это функция обработки и замены программного кода. Эта замена происходит на этапе препроцессирования, то есть на этапе обработки исходных текстов, до компиляции.

**Плюсы:** Компилятор в любом случае осуществит оптимизацию, что позволяет организовать макрос любой сложности.

**Минусы**: При работе с временными переменными необходимо определять их тип. Из-за этого макрос становится зависимым от типа переменных.

Продемонстрируем это на примере.

Пусть макрос осуществляет циклическую перестановку (a ). Сложность заключается в том, что для перестановки переменных в данном случае необходимо завести временный объект.

***Способ 1.***

Объявляем временный объект вне тела макроса, например, сделав ее глобальной или в той функции, из которой вызывается макрос.

Пример 1.

Q temp;

# define cyc\_comp (a, b, c) do { temp = a; a = b; b = c; c = temp; \

… \

} while (0);

Чтобы обезопасить код и не допустить путаницы со скобками, макрос вкладывают в цикл do … while. Обосновывается это тем, что если при компиляции обнаруживается ошибка в коде, то компилятор сообщит, что она обнаружена в цикле do … while. Это поможет быстро понять, что ошибка связана именно с макросом.

**Достоинства:** Эффективность: внутри макроса не происходит выделение памяти, благодаря этому экономится процессорное время.

**Недостатки:** Временный объект можно будет повредить в результате использования в другом месте программы.

***Способ 2.***

Объявляем временный объект в теле макроса.

Пример 2.

# define cyc\_comp (a, b, c) do { Q temp = a; a = b; b = c; c = temp; \

… \

} while (0);

**Достоинства:** Безопасность: временный объект не сможет повредиться другими функциями.

**Недостатки:** Макрос начинает зависеть от типа данных Q и каждый раз при его вызове будет выделяется память для создания временного объекта.

Нет конкретного совета, когда надо использовать способ 1, а когда – способ 2, все зависит от решения программиста и от задачи, которую он решает.

1. **Inline-подстановки**.

Чтобы организовать inline-подстановку необходимо перед типом функции добавить *inline static*.

*Static* означает, что функция исключена из таблицы экспорта. *Inline* говорит о том, что компилятор сделает эту функцию подстановкой. Если где-то происходит вызов функции, то ее тело вставится в нужное место с автоматической загрузкой нужных данных, что быстрее их выкладки на стек.

**Плюсы:** Более гибкая в использовании, в отличии от макросов.

**Минусы:** Компилятор может не сделать эту подстановку и использовать ее как обычную функцию с параметрами.

Для примера рассмотри ту же задачу, что и при использовании макроса.

***Способ 1.***

Пример 3.

inline static void cyc\_comp (Q & a, Q & b, Q &c )

{

Q temp = a; a = b; b = c; c = temp; return;

};

Здесь присутствуют те же недостатки и достоинства временного объекта, что и с макросами.

***Способ 2.***

Чтобы избавиться от привязки к типу переменных, используются шаблоны.

Пример 4.

template <class Q> inline static void cyc\_comp (Q & a, Q & b, Q & c)

{

Q temp = a; a = b; b = c; c = temp; return;

};

**Достоинства**: Временный объект не сможет быть испорчен другими функциями. Также, чтобы избавиться от привязки к типу переменных, используются шаблоны.

**Недостатки:** Выделение памяти для временного объекта каждый раз при использовании inline подстановки.

**Передача большого количества параметров**

В этом разделе также необходимо рассмотреть еще один важный вопрос – передача большого количества параметров в функцию.

Пример 5.

void my\_fun (int a, double b, char c, … и т.д.)

{

…

};

Решение простое. Необходимо данные параметры объединить в класс или структуру. Достаточно будет передать указатель на структуру или класс на вход в функцию и указать, можно ли изменять данные параметры. Для наглядности в примере 6 реализована структура, так как ее можно реализовать на С и С++ в отличии от класса.

Пример 6.

struct Q{

int a;

double b;

char c;

…

};

void my\_fun (Q & q)

{

…

};

**Достоинства:** Отсутствие явной передачи параметров через стек.

**Недостатки:** внутри функции придется пройти по указателю на передаваемую структуру или класс, что приведет к затратам на выполнение адресной арифметики и работой со структурой.

Нет явного улучшения при использовании структуры, но код выглядит более читабельно и аккуратно.

**Передача большого количества параметров**

В этом разделе также необходимо рассмотреть еще один важный вопрос – передача большого количества параметров в функцию.

Пример 5.

void my\_fun (int a, double b, char c, … и т.д.)

{

…

};

Решение простое. Необходимо данные параметры объединить в класс или структуру. Достаточно будет передать указатель на структуру или класс на вход в функцию и указать можно ли изменять данные параметры с помощью служебного слова *const*. Для наглядности в примере 6 реализована структура, так как ее можно реализовать на С и С++ в отличии от класса.

Пример 6.

struct Q{

int a;

double b;

char c;

…

};

void my\_fun (Q & q)

{

…

};

**Достоинства:** Отсутствие явной передачи параметров через стэк.

**Недостатки:** Внутри функции придется пройти по указателю на передаваемую структуру или класс, что приведет к затратам на выполнение адресной арифметики и работой со структурой.

Нет явного улучшения при использовании структуры, но код выглядит более читабельно и аккуратно.

**Как избегать замедления работы конвейера.**

В современных процессорах для организации вычислительных операций используется способ «конвейера команд». Назначение этого способа в том, чтобы ускорить работу процессора. Его идея заключается в том, чтобы распараллелить вычисление нескольких инструкций. Так как каждая ступень обработки требует своих собственных устройств, своих собственных ресурсов, которые, в свою очередь мало пересекаются, что позволяет в обработке держать несколько команд процессора, в разных стадиях выполнения, также менять их местами внутри конвейера.

Рассмотрим ситуации, когда выполнение инструкций может замедлиться.

Dependency (от англ. зависимость) - «что-то следующее зависит от предыдущего». Когда инструкция на стадии конвейера зависит от результата предыдущей стадии.

Пример:

Здесь мы видим, что для выполнения второй строки необходимо знать результат первой, также для выполнения третьей строки, необходимо знать результат и первой, и третьей. Мы может пройтись последовательно в конвейере, но тогда некоторые стадии будут ждать другие, что замедлит работу. Решением является переупорядочивание инструкций между стадиями конвейера, которые не зависят друг от друга, то есть, между вычислениями второй и третьей строки поставить команду, не зависящую от “a”.

Stall (от англ. заклинивание) Когда происходит обращение к памяти (не кэша) приходится останавливать выполнение всего конвейера. Случается, это потому, что ваша инструкция не может выполниться до конца и освободить устройство, на котором она выполняется. При этом следующая инструкция, которая не может получить доступ к этому устройству вынуждена ждать, и также со следующей за ней инструкцией, что приводит к «пробке». Решением данной проблемы является распределение между стадиями так, чтобы положить одни инструкции в кэш, и, пока они будут ждать свои данные, остальные могли выполняться.

Также есть stall, связанный с устройствами выполнения операции, например, устройство для выполнения операций с плавающей точкой, устройства, которые занимаются ветвлениями. Несмотря на то, что в процессоре может быть много ядер, количество таких устройств определенно, то есть несколько ядер разделяют между собой общее устройство, и появляется проблема, когда несколько ядер хотят использовать одно и то же устройство. Допустим, несколько стадий используют устройство с плавающей точкой, и, чтобы не создавать stall, нужно переупорядочить операции так, чтобы операции с целочисленными шли рядом с операциями с плавающей точкой, указателями и т.д.

If. Блок Prefetch, который позволяет обнаружить инструкции до их входа в конвейер. Prefetch заранее готовит данные, нужные инструкции, также заранее подбирает и подсчитывает из памяти команды нужные этой инструкции. Но при возникновении if, появляется ветвление, что приводит к необходимости делать speculative execution (умозрительное заключение). Избежать этой проблемы можно, если мы будем считать ветви условия на разных ядрах, проблема этого решения в том, что мы, как минимум, проделаем половину работы просто так (в случае двух ветвей условия), что является энергозатратным.

Однако, данное решение относится больше к сильным процессорам, а для таких CPU как Pentium M(Mobile) или ARM CPU’s, предназначенных для мобильных устройств энергозатраты будут слишком велики, что делает данное решение непригодным. В этой ситуации остается только стараться избегать If’ов.

**Виртуальная машина**

В случаях, когда необходимо запустить некое приложение, разработанное под ОС (операционную систему), отличной от установленной на рабочей машине и нет возможности/желания ставить ее параллельно, есть специальный инструмент, именуемый виртуальной машиной. Виртуальная машина (ВМ) – процессы, запущенные из хостовой (базовой) ОС, установленное на «голое железо» компьютера. Они формируют некую среду, изображающая себя «голым железом». Такая среда имеет все средства для получения для себя ресурсов: выделение памяти, доступ к графическому адаптеру, доступ к сетевому адаптеру и т.д. Для хостовой системы ВМ – всего лишь программа, а для своих внутренностей – это самое настоящее железо.

Внутрь ВМ ставится одна или несколько внутренних для нее ОС, которые называются гостевыми. Например, рядом со внешней ОС почти невозможно установить другую ОС, тогда можно на хостовой системе запустить ВМ, внутри которой можно без проблем установить две разные ОС, которые не «видят» друг друга и внешнюю машину. Иначе говоря, они полностью изолированы.

Также, в некоторых программах есть специальных режим песочницы (sandbox). Это своего рода примитивная ВМ, возможности которой сильно ограничены, но можно делать что угодно без последствий для основной системы. Такое решение используется в антивирусах для просмотра поведения потенциальной угрозы посредством ее запуска в песочнице без последствий. Отсюда следует, что хорошая ВМ должна эмулировать некое подобие жестких дисков. ВМ создает большие файлы в хостовой системе, в рамках которых изображаются жесткие диски для ОС внутри ВМ. Из них увидеть внешнюю ОС невозможно.

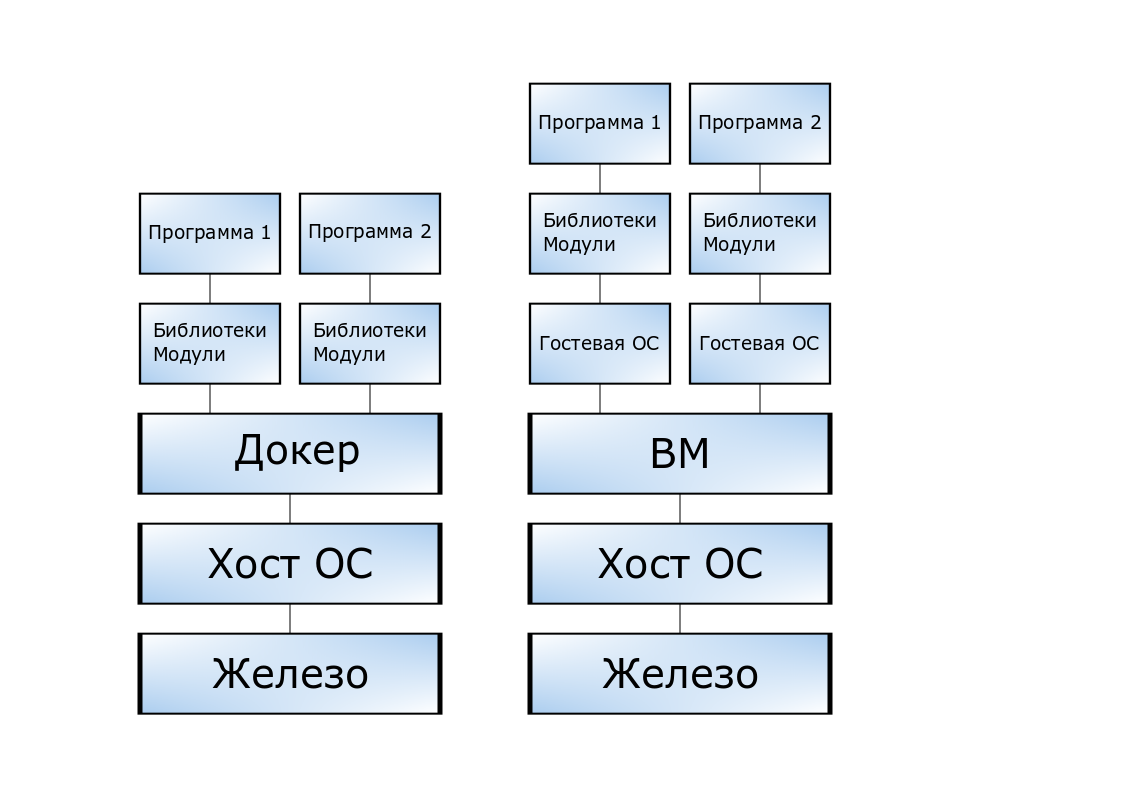
В чем отличия докера от ВМ:

1. В ней нет полной изоляции внутренней системы и внешней. Нет возможности имитировать другую архитектуру, в силу того, что программа докера использует те же самые библиотеки, ресурсы, которые предоставляет хост
2. Поскольку докер «заслоняет» собой некоторую часть окружения хостовой системы, делает некоторую часть библиотек невидимым, вместо них для программы он предлагает другой набор (например, другие версии).
3. Виртуальная машина в среднем замедляет содержимое гостевой системы до десятков процентов, потому что «прослойка» ВМ – «толстая», эмулирует каждое событие, то есть на поступающие изнутри запросы выдает ответ, изображающий что-то правдоподобное поведению оригинального железа. Это требует больших накладных расходов (процессорное время, память). В случае с докером эффективность в отношении процессорного времени близка к 100%.

Таким образом, докер – более тонкая оболочка, нежели ВМ.

Если вам нужно всего лишь поддерживать много разных сред для отладки, написания ПО, то докер лучше. Если же необходимо полностью изолировать систему при неизбежности работы с опасным ПО – лучше воспользоваться ВМ.

Контейнер – созданная в докере имитация окружения ОС + переменные среды, библиотеки. Таким образом, в докере может существовать много различных контейнеров.

Поработав с ВМ, вы можете сохранять или не сохранять результат работы. В случае с докером все, что делалось в его контейнере, не сохранится.

**Хранение или вычисление заново?**

Иногда в коде возникает необходимость вычислять одно и тоже значение от какой-нибудь функции несколько раз. Эту процедуру можно реализовать двумя способами.

1 способ.

Необходимо завести временную переменную, в которую один раз запишется вычисленный результат. И далее в коде используется уже имеющийся готовый результат.

Пример 1

double PI = 3.14;

double t = tan (45 \*PI/180);

for (int i=0; i<n;i++)

{

…

a = t;

…

}

*Применение.*

* Если таких значений много и они идут поблочно.
* Если этих значения мало и они встречаются очень часто.

2 способ.

Каждый раз вычислять заново результат функции.

Пример 2

double PI = 3.14;

for (int i=0; i<n;i++)

{

…

a = tan (45 \*PI/180);

…

}

Плюс в том, что не надо хранить в памяти временную переменную с результатом. Тяжеловесная функция вычисляется за несколько десятков тактов (~50). Адресация в памяти – порядка 100 тактов, если данный результат не попадает в кэш.

**Утечка памяти**

Утечка памяти (memory leak) — процесс неконтролируемого уменьшения объёма свободной оперативной или виртуальной памяти компьютера, связанный с ошибками в работающих программах, вовремя не освобождающих ненужные участки памяти, или с ошибками системных служб контроля памяти.

Если программист забудет выделить память под какой-то объект и что-то попробует записать, процессор определит факт, что в невыделенную область памяти пишут и вызовет исключение, которое программа вынуждена будет обработать. Обратная ситуация, если память была выделена, но не очищена после использования. Пока таких «ненужных» данных находится в памяти немного, программа может продолжать исправно работать до тех пор, пока они не заполнят лимит выделенной для приложения оперативной памяти. Это приводит к замедлению программы и всей операционной системы или полному отказу программы вследствие нехватки памяти.

Если вы запустили диспетчер задач и, наблюдая за приложением, которое простаивает, заметили, что оно потребляет все больше памяти – это верный признак, что в приложении есть ошибки такого рода.

Могу ли утечки возникнуть из-за работы со стеком? На стеке автоматически создается новый объект, когда открывается новый блок и происходит определение этого объекта и по закрытии блока объект автоматически удаляется своим деструктором. Таким образом, если все в порядке с логикой программы, всякая память, очистится, когда она не нужна, то есть, когда закрылся соответствующий блок.

Иная ситуация при работе с кучей. Утечка происходит, когда программа теряет адрес некоторой динамически выделенной части памяти, прежде чем вернуть её обратно в операционную систему. Когда это происходит, то программа уже не может удалить эту динамически выделенную память, поскольку она больше не знает, где та находится. Операционная система также не может использовать эту память, поскольку считается, что та по-прежнему используется вашей программой.

Хотя утечка памяти может возникнуть и из-за того, что указатель выходит из области видимости, возможны и другие способы, которые могут привести к утечкам памяти. Например, если указателю, хранящему адрес динамически выделенной памяти, присвоить другое значение:

int value = 7;

int \*ptr = new int; // выделяем память

ptr = &value; // старый адрес утерян - произойдет утечка памяти

Это легко решается удалением указателя перед операцией переприсваивания:

int value = 7;

int \*ptr = new int; // выделяем память

delete ptr; // возвращаем память обратно в операционную систему

ptr = &value; // переприсваиваем указателю адрес value

Кроме того, утечка памяти также может произойти и через двойное выделение памяти:

int \*ptr = new int;

ptr = new int; // старый адрес утерян - произойдет утечка памяти

Адрес, возвращаемый из второго выделения памяти, перезаписывает адрес из первого выделения. Следовательно, первое динамическое выделение становится утечкой памяти.

Чтобы найти место утечки в программе нужно очень хорошо представлять логику ее работы и постоянно следить за выделением и освобождением памяти.

Программа «» позволяет найти ошибки работы с памятью. Вместо использования стандартной библиотеки языка память заказывается через специальную виртуальную машину: она контролирует и «запоминает» куда и сколько памяти ушло и сколько возвращено в каждый момент времени работы программы.

**Треды**

Современные операционные системы и микропроцессоры давно поддерживает многозадачность и вместе с тем, каждая из этих задач может выполняться в несколько потоков. Это дает прирост производительности вычислений и позволяет лучше масштабировать пользовательские приложения и сервера, но за это — усложняется разработка программы и ее отладка.

Как минимум, в процессе должен быть хотя бы один поток (но их может быть несколько), так как в конкретном потоке происходит выполнение логики, которое описано у нас в программе. Поток определяет последовательность исполнения кода программы, даже если программа содержит функцию, либо создание какого-то класса, выполнение методов этих классов, либо цикл – все эти действия будут выполняться последовательно, согласно предусмотренной логике. С помощью создания дополнительных потоков и передачи в них части какой-то логики, например, выполнения какой-то функции, мы можем обеспечить выполнение нашего кода не последовательно, а параллельно или асинхронно, что ускорит выполнение нашей программы.

Рассмотри пример:

#include <iostream>

#include <thread>

#include <chrono>

int main()

{

setLocale(LC\_ALL, "ru");

cout << “START MAIN” << endl;

this tread::sleep\_for(chrono::milliseconds(1000));

cout << this\_thread::get(id) << endl;

cout << “END MAIN” << endl;

return 0;

}

get(id) – функция получения идентификатора потока.

sleep\_for(chrono::milliseconds) – метод, который приостанавливает работу текущего потока на определенный период времени, то есть задерживает выполнение нашего кода. Благодаря такой возможности можно симулировать длительное выполнение какой-либо сложной задачи.

Рассмотрим следующий пример:

#include <iostream>

#include <thread>

#include <chrono>

int main()

{

setLocale(LC\_ALL, "ru");

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tmain” << endl;

this tread::sleep\_for(chrono::milliseconds(3000));

}

return 0;

}

Строка с функцией sleep\_for(chrono::milliseconds(3000)) задерживает выполнение программы на 3 сек, если эту строку закомментировать, то программа выполнится мгновенно.

Для того, чтобы отследить, как это работает, что за чем выполняется, и в чем их преимущество – потребуется замедлять их выполнение.

Рассмотрим метод, который будет эмулировать сложную задачу, которая будет выполняться длительный период времени

#include <iostream>

#include <thread>

#include <chrono>

void Dowork()

{

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tDowork” << endl;

this tread::sleep\_for(chrono::milliseconds(1000));

}

}

int main()

{

setLocale(LC\_ALL, "ru");

DoWork()

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tmain” << endl;

this tread::sleep\_for(chrono::milliseconds(500));

}

return 0;

}

Как результат: сначала выполняется метод  DoWork(), а затем main. Обратим внимание, что значение id потока при выполнении этих двух методов одинаковое – это означает, что эти два метода выполнялись в одном потоке.

Если изменить sleep\_for(chrono::milliseconds(500)) на sleep\_for(chrono::milliseconds(1000)), то одна итерация метода DoWork() занимает 1 сек. Main работает немного быстрее.

Таким образом, для метода DoWork() была симулирована нагрузка.

В такой ситуации, когда есть долго выполняющаяся задача, которая блокирует работы с основным потоком, мы модем использовать дополнительный поток, чтобы распараллелить эти две задачи. Метод DoWork() можем выполнять отдельно.

#include <iostream>

#include <thread>

#include <chrono>

void Dowork()

{

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tDowork” << endl;

this tread::sleep\_for(chrono::milliseconds(500));

}

}

int main()

{

setLocale(LC\_ALL, "ru");

thread th(DoWork);

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tmain” << endl;

this tread::sleep\_for(chrono::milliseconds(500));

}

th.join();

return 0;

}

С помощью этого кода программа будет запускаться в двух потоках. Метод DoWork выполняется одновременно с методом main.

Метод th.join() позволяет дождаться выполнения той задачи, которую мы поставили в отдельном потоке.

Таким образом, задача была распараллелена.

В нескольких задачах мы можем запускать столько потоков, сколько нам нужно. Выигрыш будет в том, случае, если процессор сможет обеспечить преимущество выполнения такой задачи.