МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНОЙ РАБОТЫ №6

Лабораторная работа №6 посвящена изучению принципов работы кэш-памяти, рассмотрению понятия ассоциативности кэш-памяти и алгоритмов замещения строк кэш-памяти в случае возникновения кэш-промаха.

Кэш-память

В вычислительных системах существует известная всем проблема, заключающаяся в несоответствии скорости работы процессора и ОЗУ. Представив упрощенно работу компьютера как постоянное получение процессором данных из оперативной памяти, их изменение и сохранение обратно в ОЗУ, несложно заметить, что большую часть времени процессор в такой модели будет простаивать в ожидании завершения работы ОЗУ. Так как время работы процессора является одним из ключевых ресурсов работы системы, такое расточительство непозволительно. Оптимальным решением проблемы стало добавление при уже имеющейся большой, но медленной ОЗУ, маленькой, но быстрой памяти на микросхему процессора, которая получила название кэш-память.

Кэш-память – промежуточный буфер небольшого объема, но с быстрым доступом к нему, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Суть кэширования заключается в сокращении задержек доступа.

Оставлять кэш даже частично пустым не рационально, поэтому на всем протяжении своей работы он **всегда** будет заполнен различными данными.

Схема работы

Так как в программах преобладают обращения по чтению (из-за необходимости считывания самих команд, которые необходимо выполнить процессору), то оптимизация работы кэш-памяти заключается в ускорении работы обращений по чтению, хотя в высокопроизводительных ВС нельзя пренебрегать и скоростью операций записи.

Часто организация кэш-памяти в разных ВС отличается стратегией выполнения записи:

1. **сквозная запись** (write through), при которой измененные данные записываются сразу во все уровни иерархии памяти (разные уровни кэш-памяти и непосредственно ОЗУ);
2. **обратная запись** (write back), при которой информация записывается лишь в верхний уровень иерархии памяти, откуда при замещении строк попадает в более низкие уровни иерархии.

Сквозная запись медленнее обратной, так как происходит со скоростью самого нижнего уровня иерархии памяти (ОЗУ). При сквозной записи промахи по чтению не влияют на записи в более высокий уровень иерархии. Сквозная запись проще в реализации. Главное преимущество такого способа записи заключается в том, что

каждый из уровней иерархии имеет свежую копию данных, что крайне важно в мультипроцессорных системах (а также иногда в организации ввода-вывода).

Обратная запись быстрее сквозной, так как происходит со скоростью самого верхнего уровня иерархии памяти. При таком способе записи возможна ситуация, когда несколько записей в один и тот же блок потребует только одной записи в более низкий уровень иерархии. Также плюсом является меньшая требуемая полоса пропускания.

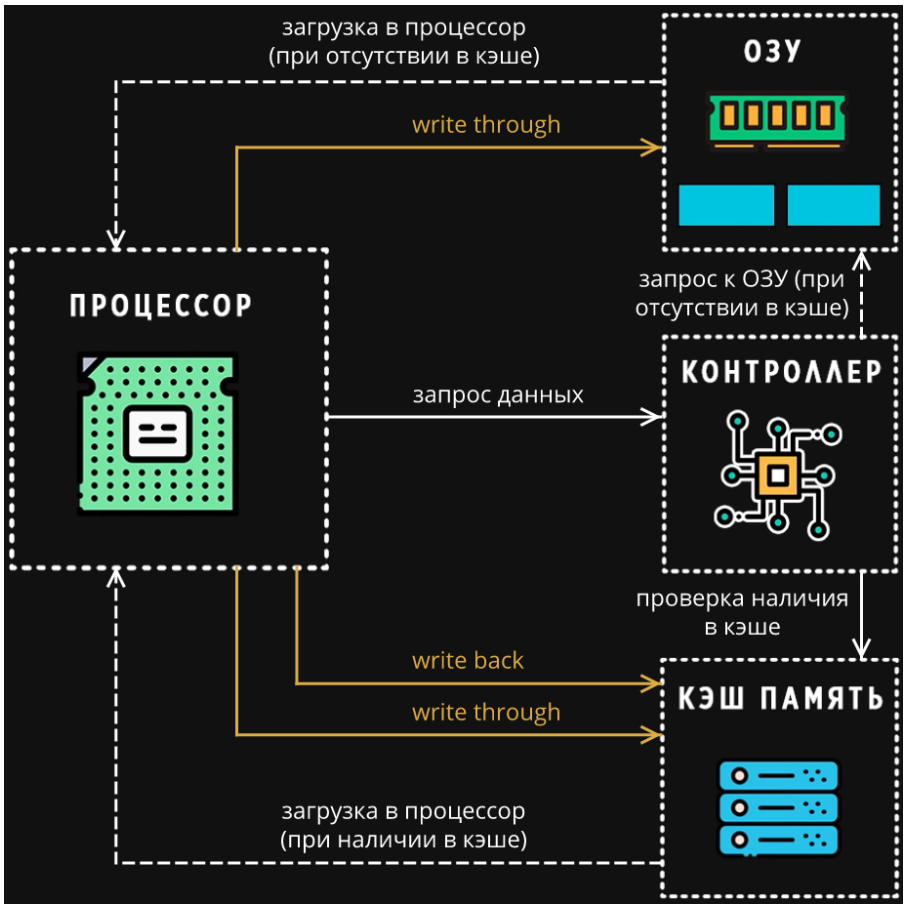

Рис. 1. Упрощенная модель работы процессора с ОЗУ

Для сохранения данных обратно в ОЗУ, необходимо знать адрес этих данных в ОЗУ, а значит необходимо их где-то хранить. Очевидным решением кажется хранение в кэше вместе с байтом данных адрес этого байта, но для этого потребовалось бы выделять большую часть объема памяти. Данные в кэше хранятся в виде набора (фиксированного размера) байтов, хранящихся в ОЗУ друг за другом. Такой набор называется **кэш-линией** (см. рисунок 2). Такая конфигурация позволяет хранить адрес не каждого байта, а лишь первого в этой кэш-линии. **Длина адреса не входит в размер кэш-линии.**

Полезный размер кэш-памяти является произведением размера кэш-линии на количество этих кэш-линий (служебные данные в этот расчет не входят). Именно полезный размер кэша указывается в документации к процессорам. **Полный размер кэш-памяти** складывается из полезного размера кэш-памяти и служебных данных к каждой из кэш-линий.

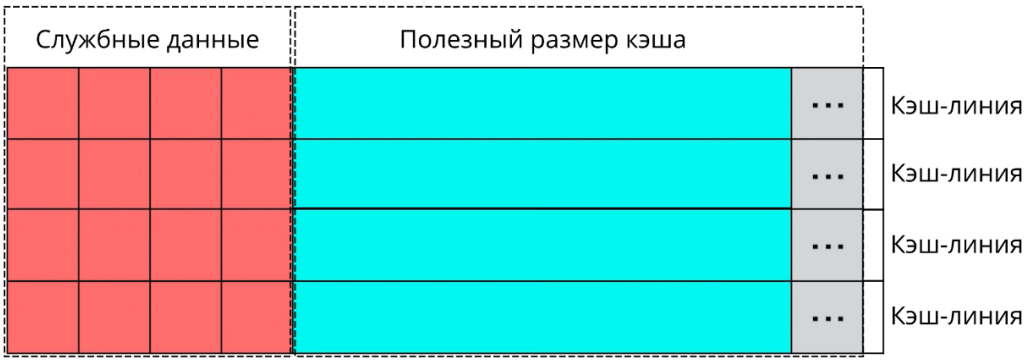

Рис. 2. Структура кэш-памяти

Кэш прямого отображения

В какую кэш-линию будет записан определенный блок данных из ОЗУ? Существуют различные способы отображения ОЗУ в кэш-память. Одним из наиболее популярных способов является прямое отображение ОЗУ в кэш-память.

Прямое отображение ОЗУ в кэш-память – способ отображения ОЗУ в кэш-память, при котором каждая строка оперативной памяти соответствует **только одной** строго определенной строке кэш-памяти, а каждой строке кэш-памяти соответствует несколько строк оперативной памяти (см. рисунок 5).

Разделив по модулю номер строки в ОЗУ на количество строк кэша, мы получим индекс кэш-строки, в которую будет помещена данная строка из оперативной памяти: $\boldsymbol{N_c \, mod \, C = i}$. Определить же в какой строке кэш-памяти будет лежать конкретный байт можно по формуле: $\frac{\boldsymbol{ADR_б}}{\boldsymbol{L}} \boldsymbol{mod \, C = i}$, где L – длина кэш-строки.

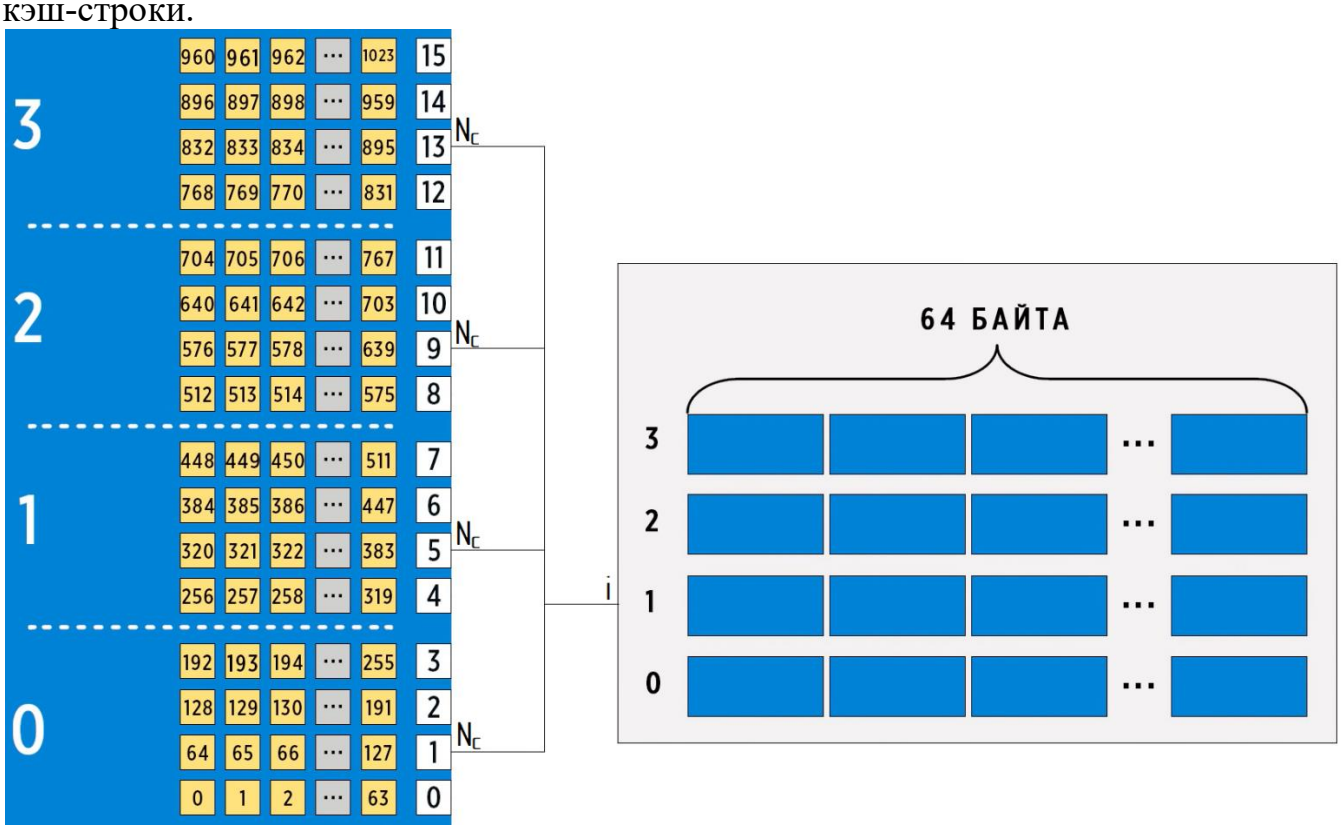

Рис. 3. Прямое отображение ОЗУ в кэш-память

Преимущество такого способа заключается в очень быстром определении нужной кэш-строки. Недостатком же является то, что на каждую строку кэш-памяти претендует сразу множество строк ОЗУ, что приводит к частому возникновению коллизий (кэш-промахов), о которых поговорим чуть позже.

Адресация данных

Рис. 4. Структура адреса данных

Длина адреса зависит от разрядности процессора и состоит из 3 частей: **тег**, **индекс строки** и **смещения** (см. рисунок 3).

Размер смещения зависит от длины кэш-строки. Например, для кэш-строки длиной 32 байта для обращения к каждому из этих байт необходимо 5 бит ($2^5 = 32$). При этом смещение также состоит из двух частей: бит, необходимых для поиска машинного слова в кэш-строке и бит для поиска конкретного байта в этом машинном слове. Например, 32-битный процессор считывает данные по 4 байта (двойное слово), значит 32-байтная кэш-строка будет содержать 8 таких двойных слов и для поиска нужного двойного слова потребуется 3 бита ($2^3 = 8$), а для поиска байта в нем – 2 бита ($2^2 = 4$).

Индекс строки зависит от количества строк в кэше – для обращения к любой из строк кэш-памяти необходимо $\boldsymbol{log_2 C}$ бит, где C – количество строк в кэш-памяти. По индексу строки контроллер кэш-памяти определяет, в какой строке кэш-памяти следует искать запрашиваемые данные.

Оставшаяся часть адреса – **тег**. Она необходима для того, чтобы определить, какая именно строка ОЗУ сейчас хранится в рассматриваемой строке кэш-памяти.

Как было описано выше, в кэше хранится адрес каждой его строки. При этом стоит отметить, что **хранить в кэше весь адрес целиком нет смысла**, так как адрес каждой строки кэш-памяти будет кратен длине строк кэш-памяти, а значит последние несколько бит всегда будут равны нулю (пример на рисунке 4). Индекс строки также не хранится в самом кэше.

Ситуация, когда в кэше были найдены запрашиваемые данные, называется **кэш-попаданием**, а ситуация, когда требующиеся данные не были найдены в кэше – **кэш-промахом**.

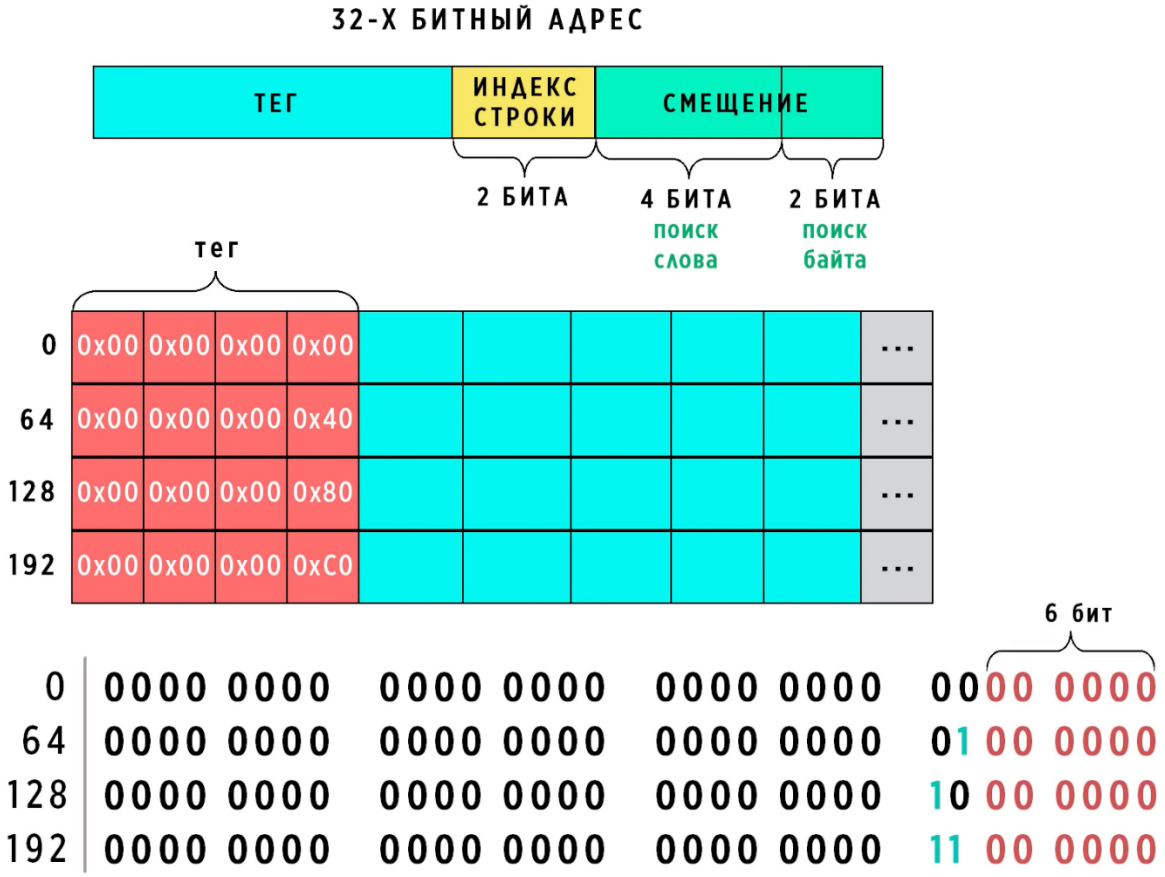

Рис. 5. Адрес кэш-строки

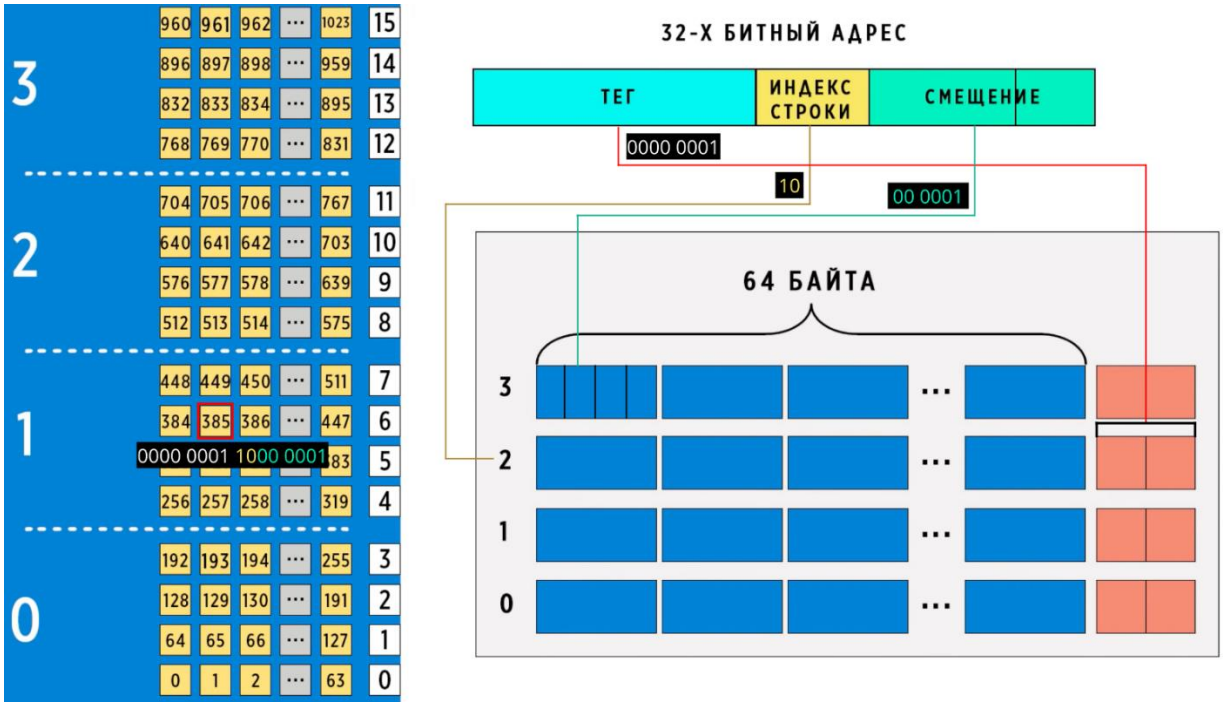

Рис. 6. Пример поиска байта в кэше

Вероятность коллизии (вероятность кэш-промаха) – вероятность, что строка кэш-памяти, в которую должны быть отображены запрашиваемые из ОЗУ данные занята другими данными из ОЗУ, также претендующими на эту кэш-линию.

В случае возникновения **коллизии** существует два возможных варианта:

1. Блок, содержащий нужный адрес, загружается в кэш-память верхнего уровня, после чего выполняется действия при записи с кэш-попаданием (обычно используется в кэш-памяти с обратным копированием);
2. Блок модифицируется на более низком уровне иерархии памяти без загрузки в верхний уровень иерархии (обычно используется в кэш-памяти со сквозной записью).

Вероятность коллизии определяется по формуле: $\boldsymbol{P}_{\text{к}} = \boldsymbol{N}_{\text{д}}/\boldsymbol{N}_{\text{отб}}$, где $N_{\text{д}}$ – количество строк в кэш-памяти, доступных для отображения строки ОЗУ, а $N_{\text{отб}}$ – количество строк ОЗУ, которые могут быть отображены в строку кэш-памяти.

Для прямого отображения при степени ассоциативности кэш-памяти равной 1 $N_{\text{д}} = 1$, а формула принимает вид: $\boldsymbol{P}_{\text{к}} = \boldsymbol{1}/\boldsymbol{N}_{\text{отб}}$.

Наборно-ассоциативный кэш

Разделим весь объем кэша на равные сегменты (количество таких сегментов должны быть кратно степени двойки) – **банки**. Каждый из этих банков представляет из себя кэш прямого отображения и является полностью ассоциативным по отношению к ОЗУ, то есть любая строка из оперативной памяти может быть размещена в любом банке кэша (но только в строго определенной для этого строке этого банка). Такое устройство кэша называется **наборно-ассоциативным**.

Степень ассоциативности кэш-памяти – количество банков, на которое разделена кэш-память (во всех ранее рассмотренных примерах степень ассоциативности кэш-памяти была равна единице, так как кэш имел один единственный банк).

Недостатком такой структуры кэша является то, что любая строчка ОЗУ может оказаться в любой из доступных ей строк кэш-памяти, а значит схема поиска данных в кэше будет сравнивать сразу несколько значений, что ведет к усложнению её реализации и, соответственно, её стоимости. Преимущество заключается в уменьшении вероятности коллизии при сохранении скорости работы.

Кэш, в котором каждая строка ОЗУ может отображаться в любую из строк кэш-памяти называется **полностью ассоциативным**. Преимущество такого кэша – низкая вероятность коллизий и высокая скорость работы, недостаток – сложная аппаратная реализация и высокая стоимость.

Алгоритмы замещения

Так как кэш-память всегда заполнена, новые данные в неё могут быть помещены лишь путем замещения старых данных. Замещение может происходить непосредственно во время обращения к кэшу, однако такой подход чреват частыми возникновениями коллизий. Чаще всего в кэше используется определенная модель предсказания того, к каким данным вскоре последует обращение и данные замещаются заранее, что повышает скорость работы. Алгоритмы, по которым определяется какие данные будут перезаписаны называются **алгоритмами замещения**.

Существует множество различных алгоритмов замещения, вот три из них:

1. **LRU** (Least Recently Used) – замещаются данные, к которым дольше всего не было обращений;
2. **MFU** (Most Frequently Used) – замещаются данные, к которым в последний раз было обращение;
3. **LFU** (Least Frequently Used) – замещаются данные, которые использовались реже всего.

Многоуровневый кэш

Современные процессоры имеют кэш нескольких уровней (как правило трех): L1, L2, L3…

L1 – самый маленький и при этом самый быстрый уровень кэш-памяти. **L2** больше по объему, но ниже по скорости. **L3** – наибольший по размеру и самый медленный (в системе трехуровневого кэша) из уровней кэша.

Каждый из старших уровней кэша может быть включающим (**inclusive**) или не включающим (**exclusive**) по отношению к младшим уровням, то есть либо содержать копии данных кэша более низкого уровня, либо не содержать таковой. Для **inclusive** архитектуры поиск данных, которые не были обнаружены в текущем уровне кэша, в кэше старших уровней искать не имеет смысла и следует обратиться напрямую к ОЗУ. Для **exclusive** архитектуры то, что данные не были обнаружены на текущем уровне кэша, не означает, что их не может быть в кэше более высокого уровня, а значит есть необходимость провести поиск данных в этих уровнях. **Exclusive** архитектура позволяет хранить больший объем данных в кэше, но может быть медленнее, чем **inclusive** архитектура.

Кэш может быть **разделяемым** (свой на каждое из ядер процессора) и **общим** (один на все ядра процессора). Общий кэш быстрее, так как в разделенном кэше, может возникнуть ситуация, когда ядру А требуется информация, просчитанная ядром Б (а значит она будет находиться в кэше ядра Б), и эта информация сперва будет помещена из кэша ядра Б в ОЗУ, а затем из ОЗУ в кэш ядра А.

Контрольные вопросы

1. Что такое кэш-память? В чем заключается суть кэширования?
2. Что такое кэш-попадание и кэш-промах? Как определить вероятность коллизии? Для какого вида отображения ОЗУ в кэш-память вероятность коллизии наименьшая?
3. Опишите процесс обратной записи (write back).
4. Приведите достоинства и недостатки различных видов отображения ОЗУ в кэш-память.
5. Что называется степенью ассоциативности кэш-памяти?
6. Перечислите известные вам алгоритмы замещения строк кэш-памяти. Когда происходит замещение строк?

ПРИЛОЖЕНИЕ

Листинг программы для выполнения ЛР6.

Для Windows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <iostream>
#include <Windows.h>
using namespace std;

#define  T char
#define  MAX_S 0x1000000
#define  L 101

volatile T A[MAX_S];
int m_rand[0xFFFFFF];

int main (){
 LARGE_INTEGER freq; LARGE_INTEGER time1;  LARGE_INTEGER time2;
  QueryPerformanceFrequency(&freq);

 memset ((void*)A, 0, sizeof (A));

 srand(time(NULL));

 int v, M;
 register int i, j, k, m, x;

 for (k = 1024; k < MAX_S;) {
 M = k / L;

 printf("%g\t", (k+M*4)/(1024.*1024));

 for (i = 0; i < M; i++) m_rand[i] = L * i;
 for (i = 0; i < M/4; i++) {
  j = rand() % M;
  x = rand() % M;

  m = m rand[j];
  m_rand[j] = m_rand[i];
  m rand[i] = m;

 }

 if (k < 100*1024) j = 1024;
 else if (k < 300*1024) j = 128;
 else j = 32;

 QueryPerformanceCounter(&time1);
 for (i = 0; i < j; i++) {

  for (m = 0; m < L; m++) {
   for (x = 0; x < M; x++){
    v = A[ m_rand[x] + m ];
   }
  }

 }
 QueryPerformanceCounter(&time2);

 time2.QuadPart -= time1.QuadPart;
 double span = (double) time2.QuadPart / freq.QuadPart;

 printf ("%g\n",1000000000. * span/(double)(L*M*j));

 if (k > 100*1024) k += k/16;
 else k += 4*1024;
 }
return 0;
}
```

Для Linux:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define  T char
#define  MAX_S 0x1000000
#define  L 101

volatile T A[MAX_S];
int m_rand[0xFFFFFF];

int main (){
 static struct timespec t1, t2;

 memset ((void*)A, 0, sizeof (A));

 srand(time(NULL));

 int v, M;
 register int i, j, k, m, x;

 for (k = 1024; k < MAX_S;) {
 M = k / L;

 printf("%g\t", (k+M*4)/(1024.*1024));

 for (i = 0; i < M; i++) m_rand[i] = L * i;
 for (i = 0; i < M/4; i++) {
  j = rand() % M;
  x = rand() % M;

  m = m rand[j];
  m_rand[j] = m_rand[i];
  m_rand[i] = m;

 }

 if (k < 100*1024) j = 1024;
 else if (k < 300*1024) j = 128;
 else j = 32;

 clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &t1);
 for (i = 0; i < j; i++) {

  for (m = 0; m < L; m++) {
   for (x = 0; x < M; x++){
    v = A[ m_rand[x] + m ];
   }
  }

 }
 clock_gettime (CLOCK_PROCESS_CPUTIME_ID, &t2);

 printf ("%g\n",1000000000. * (((t2.tv_sec + t2.tv_nsec * 1.e-9) - (t1.tv_sec + t1.tv_nsec * 1.e-
9)))/(double)(L*M*j));

 if (k > 100*1024) k += k/16;
 else k += 4*1024;
 }
return 0;
}
```