1. **ВЛИЯНИЕ КЭШ-ПАМЯТИ НА СКОРОСТЬ ДОСТУПА К ДАННЫМ**

На практике часто встречается задача обработки массивов данных. С точки зрения скорости выполнения программы, важно в каком порядке обрабатываются элементы массива, т.к. от этого будет зависеть, насколько эффективно будет работать кэш-память. Основными особенностями организации кэш-память, которые играют важную роль при обработке массивов данных, являются блочное кэширование данных и аппаратная предвыборка данных в кэш. Кроме того, существенно, умещается ли обрабатываемый массив в кэш-памяти.

* 1. **Влияние уровней кэш-памяти**

Иерархия памяти включает несколько уровней кэш-памяти разного размера и с разным временем доступа. Допустим, некоторая программа производит многократную обработку элементов массива. Если построить график зависимости среднего времени доступа к одному элементу массива от размера массива, то он должен иметь нелинейный характер. При малых размера массива, когда все данные умещаются в кэш-памяти первого уровня, время доступа к элементу будет наименьшим, и не будет меняться при увеличении размера массива. Если размер массива превысит размер кэшпамяти первого уровня, то весь массив целиком уже не сможет в нем разместиться. Поэтому при обращении к некоторым элементам массива в кэш-памяти первого уровня будут случаться кэш-промахи, и элементы будут загружаться из кэш-памяти второго уровня (или оперативной памяти). В результате, чем больше кэш-промахов происходит, тем больше будет среднее время доступа к элементу, вплоть до времени доступа к следующему уровню иерархии памяти. В результате с увеличением размера массива среднее время доступа к элементу будет ступенчато возрастать. Таким образом, анализ графика может показать, каковы объемы различных уровней кэш-памяти, имеющихся в системе.

* 1. **Влияние блочной передачи данных**

Данные из оперативной памяти в кэш-память считываются целыми блоками. Размер блока равен одной или нескольким кэш-строкам. Если элементы в массиве обрабатываются последовательно один за другим, то попытка чтения первого элемента кэш-строки вызывает копирование всего блока из медленной оперативной памяти в кэш-память. Чтение нескольких последующих элементов выполняется намного быстрее, т.к. они уже находятся в быстрой кэш-памяти.

* 1. **Влияние аппаратной предвыборки данных**

В большинстве современных микропроцессоров реализована аппаратная предвыборка данных. Она устроена таким образом, что при последовательном обходе очередные данные считываются из оперативной памяти еще до того, как к ним произошло обращение. Кэш-контроллеры с высокой вероятностью распознают последовательный обход памяти и обеспечивают эффективную предварительную загрузку данных в кэшпамять. Как следствие, вероятность кэш-промахов значительно снижается. Если же элементы массива обрабатываются в более сложном порядке, то либо кэш-контроллер его не распознает, и тогда аппаратная предвыборка работать не будет, либо может распознать неправильно, что повлечёт вытеснение ещё нужных данных из кэш-памяти и увеличение среднего времени доступа к элементу массива.

1. **ИССЛЕДОВАНИЕ ВЛИЯНИЯ КЭШ-ПАМЯТИ**

Для изучения влияния кэш-памяти на скорость доступа к данным в данной лабораторной работе требуется построить программу, выполняющую обработку данных различного размера тремя характерными способами: последовательно в сторону увеличения адресов, последовательно в сторону уменьшения адресов и в случайном порядке. Исследование времени выполнения программы в зависимости от порядка обхода и объема обрабатываемых данных позволит пронаблюдать влияние кэш-памяти.

* 1. **Перебор размеров массива**

Для определения среднего времени доступа к данным, программа должна многократно выполнять чтение элементов массива заданного размера (N) в заданном порядке. Интересующий нас диапазон изменения размеров массива определяется следующими границами:

• минимальное значение Nmin выбирается заведомо меньше, чем размер кэш-памяти данных 1-го уровня (например, 1 Кбайт – 256 элементов типа int).

• максимальное значение Nmax выбирается заведомо больше, чем размер кэш-памяти последнего уровня (например, 32 MБайта).

Программа должна перебирать размеры массива от Nmin до Nmax, для каждого конкретного N определяя среднее время доступа к элементу массива. Шаг изменения N должен быть достаточно маленьким, чтобы увидеть все существенные изгибы на графике, и достаточно большим, чтобы время выполнения теста не было слишком большим. Можно использовать переменный шаг, который растет вместе с возрастанием N.

* 1. **Выполнение обхода**

Для каждого конкретного размера массива N программа должна выполнить многократное чтение элементов массива в заданном порядке. Чтобы во время обхода не тратить время на вычисление индекса каждого следующего элемента, используется следующий приём. Значения элементов массива заполняются таким образом, чтобы сформировать односвязный циклический список, в котором значение очередного элемента представляет собой номер следующего. Обход тогда может быть выполнен циклом следующего вида:

for (k=0, i=0; i++) k = x[k];

Здесь N – размер массива, K – число обходов массива.

Таким образом, для каждого конкретного N программа должна выполнять следующие действия:

1. Заполнить значения элементов, чтобы они образовали циклический односвязный список в соответствии с требуемым порядком обхода.

2. Выполнить многократный обход массива. Замерить время обхода.

**2.3 Способы обхода элементов массива**

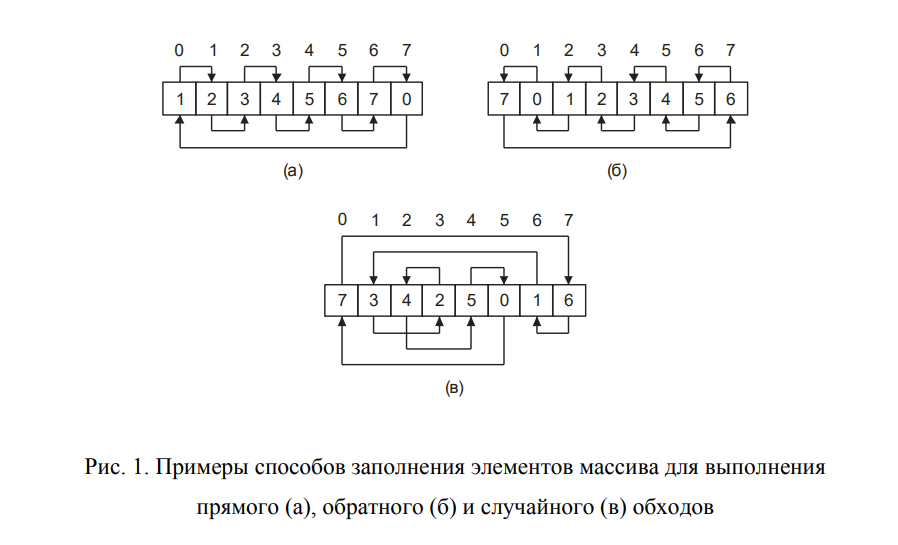
В рамках лабораторной работы предлагается сравнить время доступа к данным для трех характерных способов обхода:

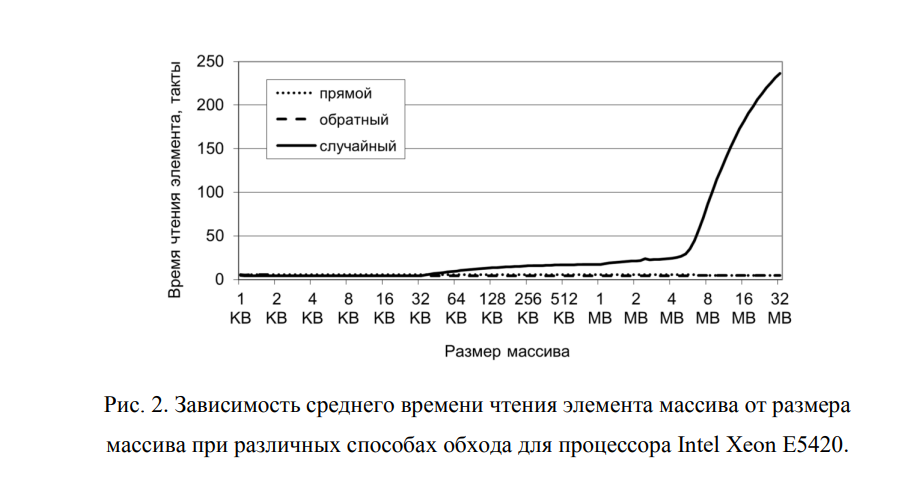
• Прямой обход – в сторону увеличения адресов,

• Обратный обход – в сторону уменьшения адресов,

• Обход элементов в случайном порядке.

Для каждого способа обхода в программе необходимо заполнить значения элементов массива таким образом, чтобы они образовали односвязный циклический список, захватывающий все элементы массива (рис. 1). Особое внимание следует обратить на заполнение массива для случайного обхода. Например, если в варианте для случайного массива, приведённом на рис. 1в, поменять местами значения ячеек 5 и 6, то в обходе будут участвовать только элементы с номерами 0, 6 и 7. Необходимо, чтобы при обходе массива участвовали все элементы.





На рис. 2 приведен пример графиков, полученных на процессоре Intel Xeon E5420 (L1: 32 KB, L2: 6 MB). Видна разница в скорости последовательного и случайного обходов массива. На соответствующих местах графика случайного обхода видно возрастание времени обращения к элементам.

**Примечание 1.** Компилировать программу нужно с ключом оптимизации –O1 чтобы исключить лишние обращения в память за счёт размещения переменных на регистрах. **Примечание 2.** Перед измерением времени для каждого размера N необходимо осуществить однократный обход массива, чтобы «прогреть кэшпамять», то есть выгрузить из кэш-памяти посторонние данные, разместив там (по возможности) необходимые нам данные. **Примечание 3.** Результирующий график может оказаться сильно "замусоренным" высокими пиками. Такое бывает, если на машине работают посторонние "тяжелые" программы. В этом случае рекомендуется встроить в программу дополнительный цикл, который бы прогонял тест несколько раз для каждого размера массива, и автоматически сохранял минимальное время по нескольким замерам.

**4. КОНТРОЛЬНЫЕ ВОПРОСЫ**

**1. Что такое кэш-память? Какую проблему она решает?**

**Кэш-память** — это быстрая, но малообъемная память, которая используется для временного хранения данных, часто используемых процессором. Она решает проблему **скорости доступа к данным**. Основная память (ОЗУ) работает медленнее, чем процессор, поэтому кэш-память позволяет сократить время ожидания процессора, когда он обращается к данным, которые уже были загружены в кэш.

**2. Какой способ обхода данных в памяти является самым быстрым? Почему?**

**Самый быстрый способ обхода данных в памяти** — это **последовательный обход** (прямой или обратный). Это связано с тем, что данные в памяти хранятся в виде блоков, и последовательный доступ позволяет использовать **локальность данных**. При последовательном обходе данные, скорее всего, уже находятся в кэше, что значительно сокращает время доступа.

**3. Какой способ обхода данных в памяти является самым медленным? Почему?**

**Самый медленный способ обхода данных в памяти** — это **случайный обход**. При случайном обходе данные не имеют локальности, и каждый доступ к памяти требует загрузки нового блока данных в кэш. Это приводит к частым промахам кэша и увеличению времени доступа к данным.

**4. Для каких программ наличие кэш-памяти дает выигрыш во времени работы?**

**Наличие кэш-памяти дает выигрыш во времени работы** для программ, которые часто обращаются к одним и тем же данным или работают с данными, имеющими высокую **пространственную и временную локальность**. Например, это типично для программ, работающих с массивами данных, циклами, итеративными алгоритмами и другими структурами, где данные повторно используются в течение короткого периода времени.

**5. Назовите две основных причины, по которым случайный обход массива дольше, чем прямой и обратный.**

**Две основные причины**, по которым случайный обход массива дольше, чем прямой и обратный:

1. **Промахи кэша**: При случайном обходе данные не имеют локальности, и каждый доступ к памяти требует загрузки нового блока данных в кэш. Это приводит к частым промахам кэша, что увеличивает время доступа.
2. **Трафик памяти**: Случайный обход генерирует больше запросов к основной памяти, что увеличивает нагрузку на шину памяти и снижает общую производительность системы.

**6. Почему тип обхода не сказывается на времени доступа к памяти, если массив умещается в кэш-памяти первого уровня?**

**Если массив умещается в кэш-памяти первого уровня**, то все данные, необходимые для обхода, уже находятся в кэше, и доступ к ним происходит очень быстро. В этом случае **тип обхода (прямой, обратный, случайный) не влияет на время доступа**, так как все данные уже загружены в кэш и доступны мгновенно.

**7. Чем отличаются программная и аппаратная предвыборка данных?**

**Программная предвыборка данных** и **аппаратная предвыборка данных** отличаются следующим:

* **Программная предвыборка**: Этот метод предполагает, что программист явно указывает, какие данные нужно загрузить в кэш заранее. Это делается с помощью специальных инструкций (например, prefetch в языках программирования). Программная предвыборка дает больший контроль над процессом, но требует от программиста знания особенностей работы кэша.
* **Аппаратная предвыборка**: В этом случае предвыборка данных осуществляется автоматически процессором на основе анализа обращений к памяти. Процессор пытается предсказать, какие данные будут нужны в ближайшее время, и загружает их в кэш. Аппаратная предвыборка прозрачна для программиста, но менее точна, чем программная.

#### 1. Написание программы

**Цель**: Написать программу, которая многократно выполняет обход массива заданного размера тремя способами: прямым, обратным и случайным.

**Шаги**:

1. **Создание массива**: Программа должна создавать массив размером N, где N — переменная, которая будет изменяться в ходе эксперимента.
2. **Обход массива**:
   * **Прямой обход**: Проход по массиву от начала до конца.
   * **Обратный обход**: Проход по массиву от конца к началу.
   * **Случайный обход**: Проход по массиву в случайном порядке. Для этого можно использовать массив индексов, который заранее перемешивается случайным образом.
3. **Измерение времени**: Для каждого способа обхода необходимо измерить время, затраченное на обход массива. Рекомендуется использовать высокопроизводительные таймеры, такие как rdtsc (Read Time-Stamp Counter) в Linux, чтобы получить время в тактах процессора.
4. **Прогрев кэша**: Перед каждым измерением времени необходимо выполнить однократный обход массива, чтобы "прогреть" кэш-память и выгрузить из нее посторонние данные.
5. **Оптимизация**: Компилировать программу с ключом оптимизации -O1, чтобы исключить лишние обращения в память за счет размещения переменных на регистрах.
6. **Повторные замеры**: Если результирующий график окажется "замусоренным", рекомендуется встроить в программу дополнительный цикл, который будет прогонять тест несколько раз для каждого размера массива и сохранять минимальное время по нескольким замерам.

#### 2. Измерение времени доступа

**Цель**: Для каждого размера массива и способа обхода измерить среднее время доступа к одному элементу (в тактах процессора). Построить графики зависимости среднего времени доступа от размера массива.

**Шаги**:

1. **Изменение размера массива**: Программа должна выполнять обходы для различных размеров массива (например, от 1000 до 1000000 элементов с шагом 1000).
2. **Вычисление среднего времени**: Для каждого размера массива и каждого способа обхода вычислить среднее время доступа к одному элементу.
3. **Построение графиков**: Построить графики зависимости среднего времени доступа от размера массива для каждого способа обхода.

#### 3. Анализ результатов

**Цель**: На основе анализа полученных графиков определить размеры кэш-памяти различных уровней и объяснить разницу во времени доступа при различных способах обхода.

**Шаги**:

1. **Определение размеров кэш-памяти**:
   * **Первый уровень (L1)**: Определить размер массива, при котором время доступа начинает резко увеличиваться. Этот размер соответствует размеру кэш-памяти L1.
   * **Второй уровень (L2)**: Аналогично определить размер массива, при котором время доступа начинает увеличиваться после заполнения L1 кэша. Этот размер соответствует размеру кэш-памяти L2.
   * **Третий уровень (L3)**: Если есть, определить размер массива, при котором время доступа начинает увеличиваться после заполнения L2 кэша.
2. **Сравнение с реальными значениями**: Сопоставить полученные значения с известными реальными значениями размеров кэш-памяти на используемом процессоре.
3. **Объяснение разницы во времени доступа**:
   * **Прямой и обратный обход**: Объяснить, почему время доступа при прямом и обратном обходе примерно одинаково и меньше, чем при случайном обходе.
   * **Случайный обход**: Объяснить, почему время доступа при случайном обходе больше, чем при прямом или обратном. Указать на промахи кэша и увеличение трафика памяти как основные причины.

**Доказать, что индeксы образуют полный цикл из всех элементов массива.**

Чтобы доказать, что индексы образуют полный цикл из всех элементов массива после выполнения функции shuffle, мы можем проанализировать алгоритм перемешивания. Функция shuffle использует алгоритм Фишера-Йетса (Fisher-Yates shuffle), который гарантирует, что каждый элемент будет перемешан случайным образом и каждое возможное перемешивание будет иметь равную вероятность.

**Алгоритм Фишера-Йетса**

Алгоритм Фишера-Йетса работает следующим образом:

1. Начиная с последнего элемента массива, выбирается случайный элемент из всех элементов, которые еще не были перемешаны.
2. Выбранный элемент меняется местами с текущим элементом.
3. Процесс повторяется для всех элементов, двигаясь справа налево.

**Доказательство**

1. **Инициализация массива индексов**:

for (int i = 0; i < MAX\_SIZE; i++) {

indices[i] = i;

}

В начале массив indices содержит индексы от 0 до MAX\_SIZE - 1 в порядке возрастания.

1. **Перемешивание массива индексов**:

void shuffle(int\* array, int n) {

for (int i = n - 1; i > 0; i--) {

int j = rand() % (i + 1);

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

* + На каждой итерации i выбирается случайный индекс j из диапазона [0, i].
  + Элементы array[i] и array[j] меняются местами.

1. **Полнота цикла**:
   * После выполнения функции shuffle, каждый индекс indices[i] будет содержать случайный индекс из диапазона [0, MAX\_SIZE - 1].
   * Так как каждый индекс изначально присутствовал в массиве и был перемешан случайным образом, каждый индекс будет присутствовать в массиве после перемешивания.
   * Это гарантирует, что каждый элемент массива будет посещен ровно один раз при случайном обходе.
2. Операция (i + 1) % size гарантирует, что после последнего элемента снова идёт первый, так как % size обнуляет индекс, если он превышает размер массива.
3. Например, если после перемешивания массив выглядит так:

arr = [3, 1, 4, 0, 2]

То после формирования цикла он будет выглядеть так:

arr = [1, 4, 0, 2, 3]

* Например, если indices = [2, 0, 1], то:
  + arr[2] = indices[1] = 0
  + arr[0] = indices[2] = 1
  + arr[1] = indices[0] = 2
* Последний элемент arr[indices[N-1]] указывает на первый элемент indices[0], что создает циклическую структуру.