**Лабораторна робота №5**

**Влияние уровней кэш-памяти на время обработки массивов данных**

**Цель работы**: изучение влияния уровней кэш-памяти на время обработки массивов данных

*Короткие теоретические сведения*

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.

1.1 Зависимость времени доступа к элементам массива от 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 кэш-памяти 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.

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.

**Завдання**

1. Написать программу, многократно выполняющую обход массива заданного размера тремя способами.

2. 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. Указать для какого процессора проводилось исследование.

3. На основе анализа полученных графиков: определить размеры кэш-памяти различных уровней, обосновать ответ, сопоставить результат с известными реальными значениями; определить размеры массива, при которых время доступа к элементу массива при случайном обходе больше, чем при прямом или обратном; объяснить причины этой разницы во временах.

**Розв’язок**

1. **Текст програми**

#include <iostream>

#include <fstream>

#include <intrin.h>

#include <thread>

#include <ctime>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

template<typename F>

unsigned long long measure(F&& f, const int n = 10);

int getSystemTime();

int\* getTargArr(int len, string type);

inline void loopTargArr(int \*arr, int n);

unsigned long long measureTargArr(int \*arr, int len);

int getRepeatVal(int len);

vector<int> getLens(int k = 1.5);

string formatSize(int bytes);

bool saveInFile(string data, string src);

void printArr(int \*arr, int len);

void run(), printHello();

template <typename T>

T prompt(const char label[]);

int main() {

   srand((int)time(0));

   printHello();

   while (1) {

      run();

      cout << endl;

      auto repeat = prompt<string>("Repeat? (1 - yes): ");

      if (repeat != "1") break;

   }

   return 0;

}

// Возвращает массив для тестирования с размером len

// Заполненный type способом в виде цикличного списка

// type = "preorder" - прямой | "postorder" - обратный | "randorder"

int\* getTargArr(int len, string type) {

   int \*arr = new int[len];

   if (type == "preorder") {

      for (int i = 0; i < len; i++) {

         arr[i] = (i + 1) % len;

      }

   }

   else if (type == "postorder") {

      for (int i = 0; i < len; i++) {

         arr[i] = (i - 1 + len) % len;

      }

   }

   else if (type == "randorder") {

      vector<int> links(len);

      for (int i = 0; i < len; i++) {

         links[i] = i;

      }

      random\_shuffle(links.begin() + 1, links.end());

      for (int i = 0; i < len; i++) {

         arr[links[i]] = links[(i + 1) % len];

      }

      /\*

      links = {0,2,1,3,4};

      arr[0] = 2;

      arr[2] = 1;

      arr[1] = 3;

      arr[3] = 4;

      arr[4] = 0;

      \*/

   }

   return arr;

}

inline void loopTargArr(int \*arr, int n) {

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

      link = arr[link];

   }

}

void printArr(int \*arr, int len) {

   for (int i = 0; i < len; i++) {

      cout << arr[i] << ' ';

   }

   cout << endl;

}

template<typename F>

unsigned long long measure(F&& f, const int n = 10) {

   unsigned long long res = UINT64\_MAX;

   for (int i = 0; i < n; i++) {

      std::this\_thread::yield();

      \_\_asm xor eax, eax

      \_\_asm cpuid

      auto start = \_\_rdtsc();

      f();

      \_\_asm xor eax, eax

      \_\_asm cpuid

      auto time = \_\_rdtsc() - start;

      if (time < res) res = time;

   }

   return res;

}

//getCPUTime

int getRepeatVal(int len) {

   if (len < 100'000) return 100;

   else if (len < 1'000'000) return 10;

   return 1;

}

unsigned long long measureTargArr(int \*arr, int len) {

   int fullLen = len;

   int repeat = getRepeatVal(len);

   // Подготовительный обход

   //loopTargArr(arr, len);

   auto time = measure([&arr, fullLen] {

      loopTargArr(arr, fullLen);

   }, repeat);

   return time / fullLen;

}

int getSystemTime() {

   const int len = 1'000'000;

   auto overhead = measure([len] {

      for (int i = 0; i < len; i++) {};

   }, 50);

   return int(overhead / len);

}

vector<int> getLens(int k = 1.5) {

   const int MIN\_LEN = 1024 / 4; //256

   const int MAX\_LEN = 5 \* 1024 \* 1024 / 4; //8388608

   vector<int> lens;

   for (int len = MIN\_LEN, step = 1; len < MAX\_LEN; len += step) {

      lens.push\_back(len);

      step = int(step \* k + 10);

   }

   lens.push\_back(MAX\_LEN);

   return lens;

}

bool saveInFile(string data, string src) {

   ofstream fout(src);

   if (!fout) return false;

   fout << data;

   fout.close();

   return true;

}

string formatSize(int bytes) {

   const int KB = 1024;

   const int MB = 1024 \* 1024;

   if (bytes < KB) {

      return to\_string(bytes) + " bytes";

   }

   float size = (float)bytes / (bytes < MB ? KB : MB);

   string strSize = to\_string(size);

   strSize = strSize.substr(0, strSize.length() - 4);

   return strSize + " " + (bytes < MB ? "KB" : "MB");

}

void printHello() {

   cout << "- - - Cash diagnostic - - -" << endl << endl;

}

template <typename T>

T prompt(const char label[]) {

   cout << label;

   while (true) {

      T val;

      cin >> val;

      if (cin.fail()) {

         cin.clear();

         cin.ignore(32767, '\n');

         cout << "Wrong. Try again: ";

      }

      else {

         cin.ignore(32767, '\n');

         return val;

      }

   }

}

void run() {

   int lenK = prompt<float>("Enter the increment step: ");

   int systemTime = getSystemTime();

   vector<int> lens = getLens(lenK);

   string res = "Length,Size,Preorder,Postorder,Randorder\n";

   cout << res;

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

      int len = lens[i];

      auto arr1 = getTargArr(len, "preorder");

      auto arr2 = getTargArr(len, "postorder");

      auto arr3 = getTargArr(len, "randorder");

      auto r1 = measureTargArr(arr1, len) - systemTime;

      auto r2 = measureTargArr(arr2, len) - systemTime;

      auto r3 = measureTargArr(arr3, len) - systemTime;

      delete[] arr1;

      delete[] arr2;

      delete[] arr3;

      string curRes = to\_string(len) + ","

         + formatSize(len \* 4) + ","

         + to\_string(r1) + ","

         + to\_string(r2) + ","

         + to\_string(r3) + "\n";

      res += curRes;

      cout << to\_string(i + 1) << "/" << to\_string(lens.size()) << "," << curRes;

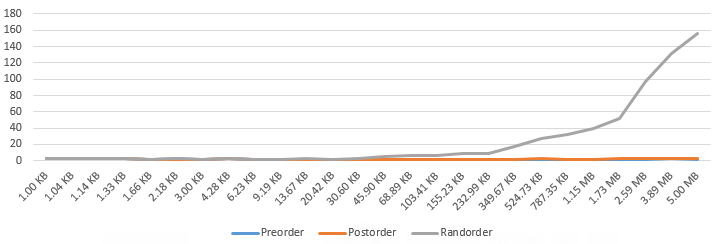
   }

   bool saved = saveInFile(res, "cash.csv");

   cout << (saved ? "Saved" : "Cannot save results to file") << endl;

}

1. **Результат виконання**



1. **Аналіз результатів**

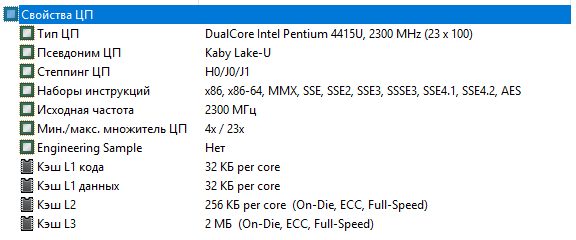
1) С графика видно, что на исследуемом процессоре имеются такие уровни кэша:

L1 ≈ 31 KB

L2 ≈ 230 KB

L3 ≈ 1.73 MB

Исследование проводилось на процессоре:



2) Размеры массива, при которых время доступа к элементу массива при случайном обходе больше, чем при прямом или обратном

Становить где-то 31 KB, так, как только в этом случае все элементы помещаются в кэш первого уровня и поэтому время доступа будет максимальным и одинаковым при любом обходе.

**Висновок:** Во время выполнения лабораторной работы я изучил влияния уровней кэш-памяти на время обработки массивов данных