Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3 по курсу**

**«Операционные системы»**

**Тема работы**

**Управление потоками в ОС и их синхронизация**

Студент: Полонский Кирилл Андреевич

Группа: М8О-208Б-20

Вариант: 10

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

<https://github.com/kirillpolonskii/OS/tree/master/os_lab3>

**Постановка задачи**

Программа получает на вход размеры матрицы, размер окна фильтра, количество его применений к матрице и саму матрицу. Далее программа в отдельном потоке, число которых не превышает заданный лимит, обрабатывает каждую строку согласно алгоритму медианной фильтрации, записывая новые элементы в новую матрицу.

**Общие сведения о программе**

Файл main.cpp содержит весь исходный код, сборка осуществляется с помощью утилиты make. В Makefile описаны флаги -fsanitize=address и -g для отслеживания ошибок и строк, в которых они возникают.

**Общий метод и алгоритм решения**

Программа принимает на вход описанные выше входные данные, после чего в цикле начинает создавать потоки. Если их число превышает заданный лимит, программа присоединяет поток с индексом i % maxThreads, после чего снова его создаёт, но уже с данными новой строки. В конце цикла осуществляется присоединение остальных потоков, если это требуется.

Программа запускается с ключом --max-threads=NUM, где NUM — максимальное количество потоков, которые может создать программа.

**Исходный код**

#include <iostream>

#include <thread>

#include <string>

#include <vector>

#include <algorithm>

**void** printMatrix(std::vector<std::vector<**int**> >& matrix){

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

**for**(**int** j = 0; j < matrix[0].size(); ++j){

std::cout << matrix[i][j] << ' ';

}

std::cout << std::endl;

}

}

**void** applyMedFilterToRow(std::vector<std::vector<**int**> >& srcMatrix, **int** i, std::vector<std::vector<**int**> >& resMatrix, **int** W){

std::cout << "thread = " << std::this\_thread::get\_id() << " is working on row = " << i << "\n";

**int** N = srcMatrix.size(), M = srcMatrix[0].size();

**if** (i < W / 2){ // if it's top rows of matrix

**int** missingRows = W / 2 - i;

**for**(**int** j = 0; j < M; ++j){

**if** (j < W / 2){ //if it's left columns

**int** missingCols = W / 2 - j;

**int** missingElems = (missingRows + missingCols) \* W - missingRows \* missingCols;

std::vector<**int**> elemRegion;

**for** (**int** k = 0; k < missingElems; ++k) elemRegion.push\_back(0);

**for** (**int** l = i - W / 2 + missingRows; l <= i + W / 2; ++l){

**for** (**int** m = j - W / 2 + missingCols; m <= j + W / 2; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full 0

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[missingElems - 1 + ((W \* W - missingElems) / 2)]; // median elem

resMatrix[i][j] = resOfFilter;

}

**else** **if** (j + W / 2 < M){ // if it's center columns

**int** missingElems = (missingRows) \* W;

std::vector<**int**> elemRegion;

**for** (**int** k = 0; k < missingElems; ++k) elemRegion.push\_back(0);

**for** (**int** l = i - W / 2 + missingRows; l <= i + W / 2; ++l){

**for** (**int** m = j - W / 2; m <= j + W / 2; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[missingElems - 1 + ((W \* W - missingElems) / 2)]; // median elem

resMatrix[i][j] = resOfFilter;

}

**else** {

**int** missingCols = (j + W / 2) % (M - 1);

**int** missingElems = (missingRows + missingCols) \* W - missingRows \* missingCols;

std::vector<**int**> elemRegion;

**for** (**int** k = 0; k < missingElems; ++k) elemRegion.push\_back(0);

**for** (**int** l = i - W / 2 + missingRows; l <= i + W / 2; ++l){

**for** (**int** m = j - W / 2; m <= j + W / 2 - missingCols; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[missingElems - 1 + ((W \* W - missingElems) / 2)]; // median elem

resMatrix[i][j] = resOfFilter;

}

} // now we ended work with row = i

}

**else** **if** (i + W / 2 < N){ // if it's center rows of matrix

**for**(**int** j = 0; j < M; ++j){

**if** (j < W / 2){ //if it's left columns

**int** missingCols = W / 2 - j;

**int** missingElems = (missingCols) \* W;

std::vector<**int**> elemRegion;

**for** (**int** k = 0; k < missingElems; ++k) elemRegion.push\_back(0);

**for** (**int** l = i - W / 2; l <= i + W / 2; ++l){

**for** (**int** m = j - W / 2 + missingCols; m <= j + W / 2; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[missingElems - 1 + ((W \* W - missingElems) / 2)]; // median elem

resMatrix[i][j] = resOfFilter;

}

**else** **if** (j + W / 2 < M){ // if it's center columns

std::vector<**int**> elemRegion;

**for** (**int** l = i - W / 2; l <= i + W / 2; ++l){

**for** (**int** m = j - W / 2; m <= j + W / 2; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[W \* W / 2]; // median elem

resMatrix[i][j] = resOfFilter;

}

**else** { // if it's right columns

**int** missingCols = (j + W / 2) % (M - 1);

**int** missingElems = (missingCols) \* W;

std::vector<**int**> elemRegion;

**for** (**int** k = 0; k < missingElems; ++k) elemRegion.push\_back(0);

**for** (**int** l = i - W / 2; l <= i + W / 2; ++l){

**for** (**int** m = j - W / 2; m <= j + W / 2 - missingCols; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[missingElems - 1 + ((W \* W - missingElems) / 2)]; // median elem

resMatrix[i][j] = resOfFilter;

}

} // now we ended work with row = i

}

**else** { // if it's bottom row of matrix

**int** missingRows = (i + W / 2) % (N - 1);

**for**(**int** j = 0; j < M; ++j){

**if** (j < W / 2){ //if it's left columns

**int** missingCols = W / 2 - j;

**int** missingElems = (missingRows + missingCols) \* W - missingRows \* missingCols;

std::vector<**int**> elemRegion;

**for** (**int** k = 0; k < missingElems; ++k) elemRegion.push\_back(0);

**for** (**int** l = i - W / 2; l <= i + W / 2 - missingRows; ++l){

**for** (**int** m = j - W / 2 + missingCols; m <= j + W / 2; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[missingElems - 1 + ((W \* W - missingElems) / 2)]; // median elem

resMatrix[i][j] = resOfFilter;

}

**else** **if** (j + W / 2 < M){ // if it's center column

**int** missingElems = (missingRows) \* W;

std::vector<**int**> elemRegion;

**for** (**int** k = 0; k < missingElems; ++k) elemRegion.push\_back(0);

**for** (**int** l = i - W / 2; l <= i + W / 2 - missingRows; ++l){

**for** (**int** m = j - W / 2; m <= j + W / 2; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[missingElems - 1 + ((W \* W - missingElems) / 2)]; // median elem

resMatrix[i][j] = resOfFilter;

}

**else** {

**int** missingCols = (j + W / 2) % (M - 1);

**int** missingElems = (missingRows + missingCols) \* W - missingRows \* missingCols;

std::vector<**int**> elemRegion;

**for** (**int** k = 0; k < missingElems; ++k) elemRegion.push\_back(0);

**for** (**int** l = i - W / 2; l <= i + W / 2 - missingRows; ++l){

**for** (**int** m = j - W / 2; m <= j + W / 2 - missingCols; ++m){

elemRegion.push\_back(srcMatrix[l][m]);

}

} // now vector with region of element is full

sort(elemRegion.begin(), elemRegion.end());

**int** resOfFilter = elemRegion[missingElems - 1 + ((W \* W - missingElems) / 2)]; // median elem

resMatrix[i][j] = resOfFilter;

}

} // now we ended work with row = i

}

std::cout << "thread = " << std::this\_thread::get\_id() << " stopped working on row = " << i << "\n";

}

**int** main(**int** argv, **char**\* argc[]){

**auto** start = std::chrono::high\_resolution\_clock::now();

**if** (argc[1] == "--help"){

std::cout << "main <KEY> -- applies median filter to the integer matrix" << "\n" <<

"KEY:" << "\n" <<

"--max-threads=M : set the maximum amount of threads - M" << "\n";

**return** 0;

}

// get the maximum amount of threads

std::string arg(argc[1]);

std::string strMaxThreads = arg.substr(14);

**int** maxThreads = std::stoi(strMaxThreads);

**int** N,M,K,W;

std::cout << "Enter size of matrix:" << "\n";

std::cin >> N >> M;

std::cout << "Enter a number of applies of median filter to matrix:" << "\n";

std::cin >> K;

std::cout << "Enter size of median filter window (it must be odd):" << "\n";

std::cin >> W;

**if** (maxThreads > N)

maxThreads = N;

std::vector<std::vector<**int**>> srcMatrix(N, std::vector<**int**> (M));

std::cout << "Enter the matrix:" << "\n";

**for**(**int** i = 0; i < N; ++i){

**for**(**int** j = 0; j < M; ++j){

std::cin >> srcMatrix[i][j];

}

}

std::vector<std::thread> threads(maxThreads);

std::vector<std::vector<**int**>> resMatrix(N, std::vector<**int**> (M));

**for** (**int** k = 0; k < K; ++k){

**int** runningThr = 0;

**for** (**int** i = 0; i < N; ++i){

**if** (runningThr < maxThreads){

threads[i] = std::thread(applyMedFilterToRow, std::ref(srcMatrix), i, std::ref(resMatrix), W);

++runningThr;

}

**else** {

threads[i % maxThreads].join();

--runningThr;

threads[i % maxThreads] = std::thread(applyMedFilterToRow, std::ref(srcMatrix), i, std::ref(resMatrix), W);

++runningThr;

}

}

**for** (**int** i = 0; i < maxThreads; ++i){

**if** (threads[i].joinable())

threads[i].join();

}

srcMatrix = resMatrix;

}

**auto** end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<**float**> duration = end - start;

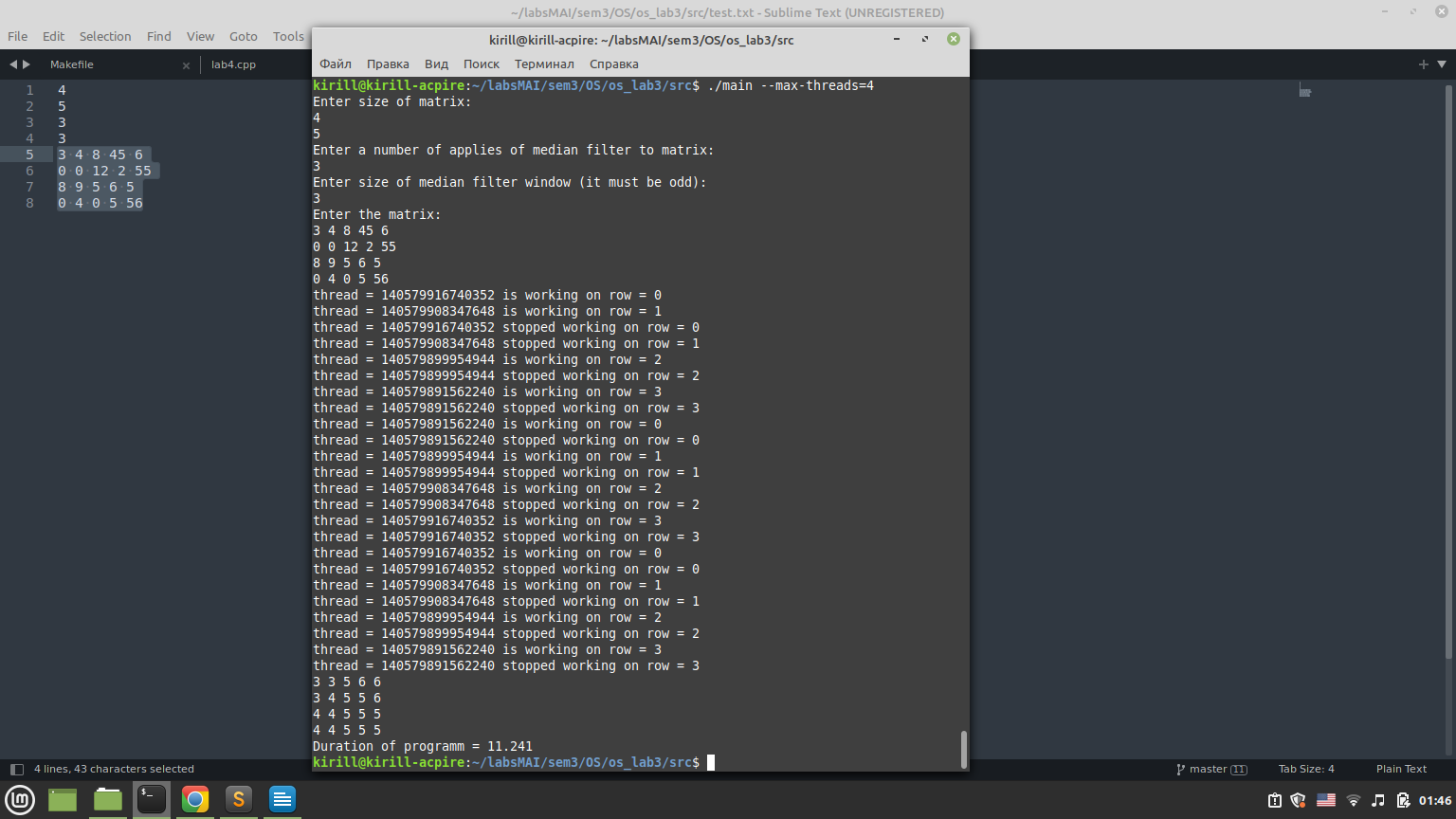
printMatrix(resMatrix);

std::cout << "Duration of programm = " << duration.count() << std::endl;

**return** 0;

}

**Демонстрация работы программы**



**Выводы**

В ходе выполнения лабораторной работы я познакомился с потоками и средствами их синхронизации, а также на практике увидел уменьшение времени работы программы при увеличении числа потоков.