Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работ №3 по курсу**

**«Операционные системы»**

Группа: М80 – 207Б – 18

Студент: Тояков Артем Олегович

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Основные файлы программы
5. Демонстрация работы программы
6. Вывод

**Постановка задачи**.

Составить программу на языке СИ, обрабатывающие данные в многопоточном режиме. При создании необходимо предусмотреть ключи, которые позволяли бы задать максимальное количество потоков, используемое программой. При возможности необходимо использовать максимальное количество возможных потоков.

Вариант 15 наложить K раз медианный фильтр на матрицу, состоящую из целых чисел, размер окна задается.

**Общие сведения о программе**

Программа компилируется из одного файла main.c. В данном файле используются заголовочные файлы stdio.h, stdlib.h, pthread.h. В программе используются следующие системные вызовы для работы с потоками из заголовочного файла pthread.h:

1. **pthread\_create –** для создания нового потока
2. **pthread\_join –** заставляет вызывающий поток ждать завершения указанного потока, используется для синхронизации потоков.

Перед написанием основной программы, было определено максимальное количество потоков в одном процессе, оно составило 2048.

Программа считывает данные с stdin, все данные подаются в виде целых чисел. В начале числа , количество раз применения фильтра, высота окна, ширина окна, количество строк, количество столбцов матрицы. Затем подаются целых чисел, элементы матрицы.

**Общий метод и алгоритм решения**.

Решение поставленной задачи делится на две части. Первая часть состоит из реализации алгоритма для одного элемента матрицы. Вторая часть состоит из использования алгоритма из первой части в многопоточном режиме.

Для того чтобы найти медианный фильтр матрицы . Необходимо применить к каждому элементу матрицы , некоторую функцию. Функция заключается в том, что по заданному окну (задается ширина и высота, в виде двух целых положительных чисел), центр которого прикладывается к текущему элементу, вычисляется медианный элемент, из множества всех элементов, попавших в заданное окно. Если не все элементы попадают в окно целиком, то необходимо за недостающие элементы взять среднее арифметическое элементов, попавших в окно. При нахождении медианного элемента окна использовался алгоритм быстрой сортировки.

После написания алгоритма для элемента , необходимо для обработки каждого элемента создавать отдельный поток. В случае, если количество потоков, которое можно создать одновременно меньше количества обрабатываемых элементов матрицы, необходимо ждать завершение потоков и создавать из снова с соответствующими элементами. Для синхронизации потоков используется вызов pthread\_join.

**Основные файлы программы**.

**Файл main.c**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define MAX\_THREAD\_NUM 2047

void swap(int \**lhs*, int \**rhs*) {

    int tmp = \*lhs;

    \*lhs = \*rhs;

    \*rhs = tmp;

}

int partition(int \**arr*, int *l*, int *r*) {

    int p = arr[(l + r) / 2];

    int i = l;

    int j = r;

    while (i <= j) {

        while (arr[i] < p)

            ++i;

        while (arr[j] > p)

            --j;

        if (i >= j)

            break;

        swap(&arr[i++], &arr[j--]);

    }

    return j;

}

void quick\_sort(int \**arr*, int *l*, int *r*) {

    if (l >= r)

        return ;

    int j = partition(arr, l, r);

    quick\_sort(arr, l, j);

    quick\_sort(arr, j + 1, r);

}

int max(int *a*, int *b*) {

    return (a > b) ? a : b;

}

int min(int *a*, int *b*) {

    return (a < b) ? a : b;

}

typedef struct {

    int win\_h;

    int win\_w;

    int mrx\_h;

    int mrx\_w;

    int \*\*i\_mrx;

    int \*\*o\_mrx;

} io\_mrx\_t;

typedef struct {

    io\_mrx\_t\* io\_mrx;

    int i;

    int j;

} thr\_data\_t;

int\*\* mrx\_create(int *h*, int *w*) {

    int \*\*arr = (int\*\*)malloc(sizeof(int\*) \* h);

    for (int i = 0; i < h; ++i) {

        arr[i]  = (int \*)malloc(sizeof(int) \* w);

    }

    return arr;

}

void mrx\_destroy(int \*\**arr*, int *h*) {

    for (int i = 0; i < h; ++i) {

        free(arr[i]);

        arr[i] = NULL;

    }

    free(arr);

}

void mrx\_input(int \*\**arr*, int *h*, int *w*) {

    for (int i = 0; i < h; ++i)

        for (int j = 0; j < w; ++j) {

            scanf("%d", &arr[i][j]);

        }

}

io\_mrx\_t\* io\_mrx\_create(void) {

    io\_mrx\_t\* io\_mrx = (io\_mrx\_t\*)malloc(sizeof(io\_mrx\_t));

    scanf("%d%d%d%d", &io\_mrx->win\_h,

        &io\_mrx->win\_w, &io\_mrx->mrx\_h, &io\_mrx->mrx\_w);

    io\_mrx->i\_mrx = mrx\_create(io\_mrx->mrx\_h, io\_mrx->mrx\_w);

    io\_mrx->o\_mrx = mrx\_create(io\_mrx->mrx\_h, io\_mrx->mrx\_w);

    mrx\_input(io\_mrx->i\_mrx, io\_mrx->mrx\_h, io\_mrx->mrx\_w);

    return io\_mrx;

}

void io\_mrx\_destroy(io\_mrx\_t\* *io\_mrx*) {

    mrx\_destroy(io\_mrx->i\_mrx, io\_mrx->mrx\_h);

    mrx\_destroy(io\_mrx->o\_mrx, io\_mrx->mrx\_h);

    io\_mrx->i\_mrx = NULL;

    io\_mrx->o\_mrx = NULL;

    free(io\_mrx);

}

void io\_mrx\_print(io\_mrx\_t\* *io\_mrx*) {

    printf("window width: %d\nwindow height: %d\nwidth: %d\nheight: %d\n",

    io\_mrx->win\_w, io\_mrx->win\_h, io\_mrx->mrx\_w, io\_mrx->mrx\_h);

    printf("\nResult matrix:\n");

    for (int i = 0; i < io\_mrx->mrx\_h; ++i) {

        for (int j = 0; j < io\_mrx->mrx\_w; ++j) {

            printf("%5d", io\_mrx->o\_mrx[i][j]);

            printf(j == io\_mrx->mrx\_w - 1 ? "\n" : " ");

        }

    }

}

thr\_data\_t\* thr\_data\_arr\_create(io\_mrx\_t\* *io\_mrx*) {

    int n = io\_mrx->mrx\_h \* io\_mrx->mrx\_w;

    thr\_data\_t\* arr = (thr\_data\_t\*)malloc(sizeof(thr\_data\_t) \* n);

    for (int i = 0; i < n; ++i) {

        arr[i].io\_mrx = io\_mrx;

        arr[i].i = i / io\_mrx->mrx\_h;

        arr[i].j = i % io\_mrx->mrx\_w;

    }

    return arr;

}

void thr\_data\_arr\_destroy(thr\_data\_t\* *arr*) {

    free(arr);

}

int arithm\_mean(int \**arr*, int *size*) {

    int res = 0;

    for (int i = 0; i < size; ++i)

        res += arr[i];

    return res / size;

}

int median\_val(int \**arr*, int *size*) {

    quick\_sort(arr, 0, size - 1);

    return (size % 2) ? (arr[size / 2 - 1] + arr[size / 2]) / 2 : arr[size / 2];

}

void item\_filter(thr\_data\_t\* *item*) {

    int i = max(0, item->i - item->io\_mrx->win\_w / 2);

    int j = max(0, item->j - item->io\_mrx->win\_h / 2);

    int end\_i = min(item->io\_mrx->mrx\_w,

        item->io\_mrx->win\_w - item->io\_mrx->win\_w / 2 + item->i);

    int end\_j = min(item->io\_mrx->mrx\_h,

        item->io\_mrx->win\_h - item->io\_mrx->win\_h / 2 + item->j);

    int count = 0;

    int size = item->io\_mrx->mrx\_h \* item->io\_mrx->mrx\_w;

    int win\_arr[size];

    for (; i < end\_i; ++i)

        for (; j < end\_j; ++j)

            win\_arr[count++] = item->io\_mrx->i\_mrx[i][j];

    if (count != size) {

        int ar\_mean = arithm\_mean(win\_arr, count);

        for (int i = count; i < size; ++i)

            win\_arr[i] = ar\_mean;

    }

    item->io\_mrx->o\_mrx[item->i][item->j] = median\_val(win\_arr, size);

}

void\* routine(void \**arg*) {

    thr\_data\_t\* thr\_data = (thr\_data\_t \*)arg;

    item\_filter(thr\_data);

    return NULL;

}

int main(int *argc*, char \*\**argv*)

{

    if (argc != 2) {

        printf("Using ./filter num\_threads\n");

        return 0;

    }

    int n = atoi(argv[1]);

    int k;

    scanf("%d", &k);

    io\_mrx\_t\* io\_mrx = io\_mrx\_create();

    thr\_data\_t\* thr\_data\_arr = thr\_data\_arr\_create(io\_mrx);

    int size = io\_mrx->mrx\_h \* io\_mrx->mrx\_w;

    n = min(max(n, 1), MAX\_THREAD\_NUM);

    pthread\_t\* threads = (pthread\_t\*)malloc(sizeof(pthread\_t) \* n);

    while (k > 0) {

        int i = 0;

        while (i < size) {

            int t = min(size - i, n);

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

                pthread\_create(&threads[j], NULL, routine, &thr\_data\_arr[i++]);

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

                pthread\_join(threads[j], NULL);

        }

        for (int l = 0; l < io\_mrx->mrx\_h; ++l)

            for (int j = 0; j < io\_mrx->mrx\_w; ++j)

                io\_mrx->i\_mrx[l][j] = io\_mrx->o\_mrx[l][j];

        --k;

    }

    io\_mrx\_print(io\_mrx);

    free(threads);

    thr\_data\_arr\_destroy(thr\_data\_arr);

    io\_mrx\_destroy(io\_mrx);

}

**Демонстрация работы программы.**

artoy@artoy:~/Desktop/Labs/3 sem/OS/Lab3$ ./filter 4

2

3

3

10

10

32 13 89 -45 -21 -54 4 62 91 66

-94 11 30 25 45 67 45 6 -32 54

-33 37 64 96 -98 -26 -45 -92 55 -29

60 7 -69 -88 -98 -62 -50 24 -94 85

83 4 93 79 -74 74 -30 -33 60 84

96 17 -15 -47 26 41 63 -15 -66 11

-12 -50 85 -41 -62 -44 9 16 -58 -44

-88 -64 -57 72 -57 93 -52 -14 73 -83

20 96 -51 15 83 34 70 43 -46 98

-11 50 -74 88 -28 82 51 -41 65 68

Number of threads: 4

Number of filter: 2

window width: 3

window height: 3

width: 10

height: 10

Result matrix:

-31 -31 -32 12 61 7 -4 -10 14 20

-28 -31 -33 12 59 25 -21 -26 4 15

-5 -22 -33 11 64 25 -98 -26 -45 -7

31 36 60 7 30 25 -98 -26 -45 -14

56 55 83 7 64 -41 -74 41 -30 -21

44 55 83 4 -15 -41 -62 41 9 -14

9 -1 -12 -50 -15 -41 -57 41 9 -4

-17 -26 -12 -50 -51 15 -57 41 9 -4

-16 -26 -11 50 -57 51 -28 58 51 1

-11 -11 -3 61 -59 61 0 70 55 -6

artoy@artoy:~/Desktop/Labs/3 sem/OS/Lab3$ ./filter 100

2

3

3

3

3

39 80 -61

-65 -89 94

-96 38 -5

Number of threads: 9

Number of filter: 2

window width: 3

window height: 3

width: 3

height: 3

Result matrix:

-26 -26 -39

-44 -40 -52

-60 -60 -72

**Вывод**

Большое отличие потоков от процессов состоит в том, что потоки делят между собой одно адресное пространство. В данной лабораторной работе была продемонстрирована обработка матрицы по определенному алгоритму в многопоточном режиме. И если посмотреть на время при работе программы при разном количестве потоков, то оно действительно различается. Обычно параллельные алгоритмы хорошо обрабатывать в многопоточном режиме на графических процессорах, так как на них очень много ядер и на них действительно можно быстро выполнять многие задачи, например обучать нейросети. Эта лабораторная работа помогла мне разобраться с многопоточной реализацией программ.