Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3 по курсу**

**«Операционные системы»**

**ПОТОКИ**

Студент: Киреев Александр Константинович

Группа: М8О–206Б–20

Вариант: 12

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2020.

**Постановка задачи**

## Цель работы

Целью является приобретение практических навыков в:

* Управление потоками в ОС
* Обеспечение синхронизации между потоками

## Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение потоков должно быть задано ключом запуска вашей программы.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

В отчете привести исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант 12:

Наложить K раз фильтр, использующий матрицу свертки, на матрицу, состоящую из вещественных чисел. Размер окна 3х3

**Общие сведения о программе**

Программа компилируется из файла lab3.c. Для компиляции файла lab3.c необходимы также файлы matrixio.c, matrixio.h. В программе используются заголовочные файлы: stdio.h, unistd.h, stdlib.h, ctype.h, stdbool.h, pthread.h, time.h, dirent.h, sys/time.h, matrixio.h. В программе используются следующие системные вызовы:

1. **gettimeofday** ­­–– позволяет получить текущее время. Принимает в качестве аргументов структуру timeval, в которую записывает результат работы, и переменную для коррекции времени. В случае успеха возвращает 0.
2. **pthread\_create ––** (является оберткой над системным вызовом clone) создает новый поток в вызывающем процессе. В качестве аргументов принимает указатель на структуру-идентификатор потока pthread\_t, атрибуты потока, функцию, которая будет запускаться в потоке, список аргументов для функции в виде указателя на void. В случае успеха возвращает 0, иначе возвращает номер ошибки.
3. **pthread\_join ––** используется для ожидания завершения потока. Данная функция блокирует вызывающий поток, пока указанный поток не завершится. В качестве аргументов принимает структуру pthread\_t потока и указатель на переменную, в которую будет записан результат, возвращаемый потоком. В случае успеха возвращает 0, иначе возвращает номер ошибки.
4. **pthread\_exit ––** завершает вызываемый поток. В качестве аргумента принимает значение, которое вернется при завершении потока. Функция всегда завершается успехом.
5. **read ––** предназначена для чтения какого-то числа байт из файла, принимает в качестве аргументов файловый дескриптор, буфер, в который будут записаны данные и число байт. В случае успеха вернет число прочитанных байт, иначе -1.
6. **write ––** предназначена для записи какого-то числа байт в файл, принимает в качестве аргументов файловый дескриптор, буфер, из которого будут считаны данные для записи и число байт. В случае успеха вернет число записанных байт, иначе -1.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы потоков в C.
2. Написать парсер для типа матриц типа float, и значений типа float, int. (учитывая не валидные данные).
3. Реализовать вспомогательные функции для логирования.
4. Разбить глобальную задачу на множество более мелких, реализовать алгоритм обработки частичных данных, который будет запускаться мультипоточно.
5. Реализовать функцию, которая будет создавать потоки, запускать функцию-обработчик и управлять потокам.
6. Реализовать обработку системных ошибок согласно заданию.

**Основные файлы программы**

**matrixio.h:**

#pragma once

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <stdbool.h>

#include <unistd.h>

#define RF\_VALID 0

#define RF\_INVALID 1

#define RF\_VALID\_ENDL 2

#define RF\_INVALID\_ENDL 3

#define RF\_VALID\_EOF 4

#define RF\_INVALID\_EOF 5

#define RI\_VALID 0

#define RI\_INVALID 1

#define RI\_EOF 2

#define STDIN 0

#define STDOUT 1

#define STDERR 2

void skip\_tail();

bool check\_tail();

bool is\_valid(char\* buf);

int read\_float(float\* n);

int read\_int(int\* n);

int scan\_matrix(int str\_c, int col\_c, float\* m);

void write\_float(float n);

void print\_matrix(int str\_c, int col\_c, float\* m);

**lab3.c:**

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <stdbool.h>

#include <unistd.h>

#include <pthread.h>

#include <time.h>

#include <dirent.h>

#include <sys/time.h>

#include "matrixio.h"

// !для удобства будем хранить матрицу, как одномерный массив длины n\*m!

long long current\_timestamp() {

struct timeval te;

gettimeofday(&te, NULL); // get current time

long long milliseconds = te.tv\_sec \* 1000LL + te.tv\_usec / 1000; // calculate milliseconds

// printf("milliseconds: %lld\n", milliseconds);

return milliseconds;

}

// структура для передачи в функцию, которая будет запущена на нескольких потоках

typedef struct container container;

struct container {

int s\_size;

int c\_size;

float\* m;

float\* w;

float\* res;

int thread\_id;

int threads;

};

// функция для вывода сообщений на экран

void logs(char\* str, int fd) {

int i = 0;

while (str[i] != '\0') {

write(fd, &str[i++], 1);

}

}

// прочитать размер матрицы и проверить на валидность

int read\_dimension(int\* d) {

if (read\_int(d) != RI\_VALID || (\*d) <= 0) {

char\* err = "EOF or invalid dimension input!\n";

logs(err, STDERR);

return RI\_INVALID;

}

return RI\_VALID;

}

// полностью скопировать матрицу такого же размера

void deep\_copy(int s\_size, int c\_size, float\* from, float\* to) {

for (int i = 0; i < s\_size; ++i) {

for (int j = 0; j < c\_size; ++j) {

to[i \* c\_size + j] = from[i \* c\_size + j];

}

}

}

// считать матрицу float и обработать ошибки

int get\_matrix(int s\_size, int c\_size, float\* m) {

int scan\_return = scan\_matrix(s\_size, c\_size, m);

if (scan\_return == RF\_INVALID || scan\_return == RF\_INVALID\_EOF) {

char endl = '\n';

char\* err = "Invalid matrix, try to enter again!\n";

if (scan\_return == RF\_INVALID\_EOF) {

write(STDERR, &endl, 1);

}

logs(err, STDERR);

} else if (scan\_return == RF\_VALID\_EOF) {

char\* err = "\nUnexpected EOF, try to enter again!\n";

logs(err, STDERR);

}

return scan\_return;

}

// считаем кусок матрицы (по строкам) по алгоритму свертки (эта функция и будет запущена на потоках)

void\* cell\_algo(void\* arg) {

// парсим аргументы

container\* c = arg;

int thread\_id = c->thread\_id, threads = c->threads,

s\_size = c->s\_size, c\_size = c->c\_size;

float\* m = c->m;

float\* w = c->w;

float\* res = c->res;

// массив для того, чтобы задавать, как нужно сместиться относительно центральной ячейки

int dir[3] = {-1, 0, 1};

// сам алгоритм (чтобы не допустить обращения тредов к одному и тому же месту памяти, будем давать им непересекающиеся секторы)

// например: тред №1 будет обрабатывать 0-ую строку, тред №2 - 1-ую строку и тд

for (int str\_c = thread\_id; str\_c < s\_size; str\_c += threads) {

for (int col\_c = 0; col\_c < c\_size; col\_c++) {

int count = 0;

float new\_val = 0.0;

// применяем матрицу свертки, обрабатывая невалидные индексы

for (int i = 0; i < 3; ++i) {

for (int j = 0; j < 3; ++j) {

if (((str\_c + dir[i]) \* c\_size + col\_c + dir[j] >= 0) &&

((str\_c + dir[i]) \* c\_size + col\_c + dir[j] < c\_size \* s\_size)) {

new\_val += w[i \* 3 + j] \* m[(str\_c + dir[i]) \* c\_size + col\_c + dir[j]];

count++;

}

}

}

new\_val /= count;

// записываем новое значение в массив

res[str\_c \* c\_size + col\_c] = new\_val;

}

}

// заканчиваем тред

pthread\_exit(NULL);

}

// алгоритм, запускающий треды

void algo(int s\_size, int c\_size, float\* m, float\* w, float\* res, int K, int threads) {

// создаем копию исходной матрицы, чтобы не испортить ее

float\* tmp\_m = (float\*)malloc(s\_size \* c\_size \* sizeof(float));

deep\_copy(s\_size, c\_size, m, tmp\_m);

// массив в котором будут храниться структуры, описывающие треды

pthread\_t tids[threads];

// нужно хранить передаваемые аргументы, иначе цикл отработает и переменная уровня цикла удалится

container c[threads];

// K - число раз, сколько нужно применить матрицу свертки

for (int \_k = 0; \_k < K; ++\_k) {

for (int i = 0; i < threads; ++i) {

// заполняем структурку

c[i].s\_size = s\_size;

c[i].c\_size = c\_size;

c[i].m = tmp\_m;

c[i].w = w;

c[i].res = res;

c[i].thread\_id = i;

c[i].threads = threads;

// создаем тред с дефолтными атрибутами, в котором будет выполнятся функция cell\_algo

if (pthread\_create(&tids[i], NULL, cell\_algo, &c[i]) != 0) {

char\* err = "Unable to create a thread!\n";

logs(err, STDERR);

}

}

// считаем число тредов (к сожалению этот кусок кода работает только на линуксе, мак не знает о /proc/)

/\* раскоментируйте этот код, если хотите получить информацию о числе тредов

if (\_k == 0) {

DIR\* dir;

struct dirent\* entry;

int pid = getpid();

char dirname[256];

sprintf(dirname, "/proc/%d/task", pid);

int threads\_c = -3;

if ((dir = opendir(dirname)) == NULL) {

char\* err = "Unable to open dir\n";

logs(err, STDERR);

} else {

while ((entry = readdir(dir)) != NULL) {

threads\_c++;

}

closedir(dir);

}

printf("Number of threads per one K: %d\n", threads\_c);

}

\*/

// ждем все треды, тк нам нужна результирующая матрица полностью, чтобы применить еще k-1 раз матрицу свертки

for (int i = 0; i < threads; ++i) {

if (pthread\_join(tids[i], NULL) != 0) {

char\* err = "Unable to wait a thread!\n";

logs(err, STDERR);

}

}

// если K > 1, то нужно перезаписать промежуточный результат во временную матрицу, чтобы работать с промежуточным результатом

if (K > 1) {

deep\_copy(s\_size, c\_size, res, tmp\_m);

}

}

free(tmp\_m);

}

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

// обрабатываем ключи

if (argc <= 2 && argc >= 4) {

char\* err = "Usage: ./executable -count\_of\_threads [-w].\n";

logs(err, STDERR);

return 1;

}

// конвертация ключа в инт, чтобы получить число тредов

char\* e;

int threads = strtol(argv[1] + 1, &e, 10);

if (threads <= 0) {

char\* err = "Count of threads must be positive.\n";

logs(err, STDERR);

return 1;

}

// ввод всех данных + обработка ошибок

char\* input\_msg = "Enter matrix dimensions, matrix, enter window, enter K.\n";

logs(input\_msg, STDOUT);

int m\_s\_size = 0, m\_c\_size = 0, w\_s\_size = 3, w\_c\_size = 3, K = 0;

// ввод матрицы исходной

if (read\_dimension(&m\_s\_size) != RI\_VALID) {

return 1;

}

if (read\_dimension(&m\_c\_size) != RI\_VALID) {

return 1;

}

float\* m = (float\*)malloc(m\_s\_size \* m\_c\_size \* sizeof(float));

if (m == NULL) {

char\* err = "Matrix m: malloc error!\n";

logs(err, STDERR);

return 1;

}

float\* result = (float\*)malloc(m\_s\_size \* m\_c\_size \* sizeof(float));

if (result == NULL) {

free(m);

char\* err = "Matrix result: malloc error!\n";

logs(err, STDERR);

return 1;

}

int get\_return = get\_matrix(m\_s\_size, m\_c\_size, m);

if (get\_return != RF\_VALID) {

free(m);

free(result);

return 1;

}

// ввод окна

float\* w = (float\*)malloc(w\_s\_size \* w\_c\_size \* sizeof(float));

get\_return = get\_matrix(w\_s\_size, w\_c\_size, w);

if (get\_return != RF\_VALID) {

free(m);

free(result);

free(w);

return 1;

}

// ввод K;

get\_return = read\_int(&K);

if (get\_return == RI\_INVALID) {

char\* err = "Invalid int input!\n";

logs(err, STDERR);

free(m);

free(result);

free(w);

return 1;

}

// запуск алгоритма и бенчмарк

long long begin, end;

begin = current\_timestamp();

algo(m\_s\_size, m\_c\_size, m, w, result, K, threads);

end = current\_timestamp();

long long time\_spent = end - begin;

if (get\_return == RI\_EOF) {

char endl = '\n';

write(STDIN, &endl, 1);

}

// вывод

if (argc != 3) {

char\* output\_msg = "Result matrix is:\n";

logs(output\_msg, STDOUT);

print\_matrix(m\_s\_size, m\_c\_size, result);

}

char\* time\_msg = "Time spent on algo (ms):\n";

logs(time\_msg, STDOUT);

printf("%lld\n", time\_spent);

free(m);

free(result);

free(w);

return 0;

}

**Пример работы**

Тестирование на больших и маленьких случайных тестах:

MacBook-Air-K:test AK$ ./wrapper.sh

[info] [2020-10-20 22:34:03] Stage #1 Compiling...

gcc -pedantic -Wall -Werror -std=c11 lab3.c matrixio.c -o lab3 -lpthread

[info] [2020-10-20 22:34:04] Stage #2 Test generating...

[info] [2020-10-20 22:34:13] Stage #3 Checking (only spent time output)...

>>Checking test\_1\_1.txt, threads: 1...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

0

>>Checking test\_1\_2.txt, threads: 1...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

1

>>Checking test\_2\_1.txt, threads: 1...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

119

>>Checking test\_2\_2.txt, threads: 1...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

226

>>Checking test\_1\_1.txt, threads: 2...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

0

>>Checking test\_1\_2.txt, threads: 2...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

0

>>Checking test\_2\_1.txt, threads: 2...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

68

>>Checking test\_2\_2.txt, threads: 2...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

138

>>Checking test\_1\_1.txt, threads: 4...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

0

>>Checking test\_1\_2.txt, threads: 4...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

1

>>Checking test\_2\_1.txt, threads: 4...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

71

>>Checking test\_2\_2.txt, threads: 4...

Enter matrix dimensions, matrix, enter window, enter K.

Time spent on algo (ms):

130

Наглядный тест:

MacBook-Air-K:test AK$ ./../src/lab3 -16

Enter matrix dimensions, matrix, enter window, enter K.

4 4

2 2 2 2

2 2 2 2

2 2 2 2

2 2 2 2

0.5 0.5 0.5

0.5 0.5 0.5

0.5 0.5 0.5

2

Result matrix is:

0.500 0.500 0.500 0.500

0.500 0.500 0.500 0.500

0.500 0.500 0.500 0.500

0.500 0.500 0.500 0.500

Time spent on algo (ms):

2

**Вывод**

В ходе данной лабораторной работы я познакомился с потоками в языке C. Потоки очень полезны в случаях, когда большую задачу можно разбить на множество более мелких задач, которые могут выполняться параллельно. Особенно многопоточное программирование начало развиваться с появлением многоядерных процессоров. Однако такое распараллеливание задачи также требует переработки стандартного алгоритма, большей внимательности к совместно-используемым данным, необходимости организовывать общение поток между собой. Исходя из замеров времени работы алгоритма на маленьких случайных данных, можно сделать вывод, что потоки – излишество на меленьких данных, так ка они не дают почти никакого прироста в производительности, а наоборот могут замедлить код, так как создание потоков и их ожидание, переключение контекста тоже занимает какое-то время. На больших случайных данных потоки дают некоторый выигрыш: примерно в два раза между однопоточной программой и двухпоточной программой, и совсем небольшой между двухпоточной программой и четырехпоточной программой. Уменьшение прироста производительности можно объяснить законом Амдала.