Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу
«Операционные системы»

УПРАВЛЕНИЕ ПОТОКАМИ В ОС И ОБЕСПЕЧЕНИЕ СИНХРОНИЗАЦИИ МЕЖДУ НИМИ

Студент: Суханов Е.А.
Группа: М8О–206Б–19
Вариант: 6
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021.

Постановка задачи

Цель работы

Приобретение практических навыков в

- Управление потоками в ОС;
- Обеспечение синхронизации между потоками.

Задание

Требуется составить многопоточную программу на языке Си, которая:

- Производит перемножение 2-ух матриц, содержащих комплексные числа;
- Количество потоков указывается в виде аргумента запуска программы.

Общие сведения о программе

Для создания многопоточной программы нужно использовать заголовочный файл pthread. Также необходимо указать линковщику ключ -pthread.

Исходный код включает следующие файлы:

- main.c - содержит точку входа, общий алгоритм;
- matrix.c, matrix.h - определяют тип данных “Matrix”, а также функции для работы с ним, в том числе многопоточное перемножение;
- io.c, io.h - так как запрещено использовать абстракции над системными вызовами ввода и вывода, здесь реализованы более высокоуровневые функции ввода-вывода;

Основные функции thread, которые я использовал:

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg), где thread - указатель для хранения идентификатора созданного потока, attr - атрибуты потока, start - функция, которая запустится в новом потоке, arg - аргументы для потоковой функции. Данная функция вернет ненулевое значение, если произошла какая-то ошибка, иначе - 0. Данная функция создает и запускает поток;
- int pthread_join (pthread_t THREAD_ID, void ** DATA), ожидает завершения потока с идентификатором THREAD_ID, DATA указывает, куда нужно записать возвращаемое значение, если он равен NULL, то

возвращаемое значение игнорируется. Функция возвращает 0, если выполнилась успешно, ненулевое значение - возникла ошибка.

Общий метод и алгоритм решения.

Алгоритм многопоточного умножения матриц:

1. Распределить нагрузку по потокам. Я решил распределять строки итоговой матрицы;
2. Для каждого потока найти строчки, которые он будет вычислять. Запустить поток;
3. Дождаться завершения каждого потока.

Исследование ускорения и эффективности алгоритма от входящих данных и кол-ва потоков

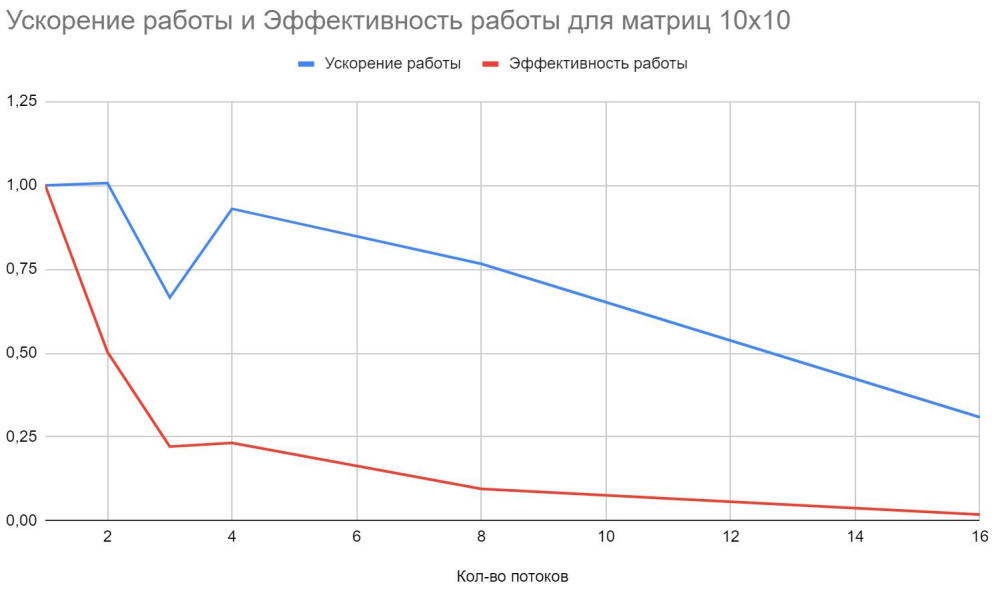
Сразу следует сказать, так как я распределяю строчки целиком, то при малом количестве строк, но при большом кол-ве столбцов, мой алгоритм будет показывать плохие результаты.

Si - ускорение времени работы на i потоках относительно времени работы однопоточного решения. Si = T1/Ti

Ei - эффективность работы i потоков. Ei = Si / i

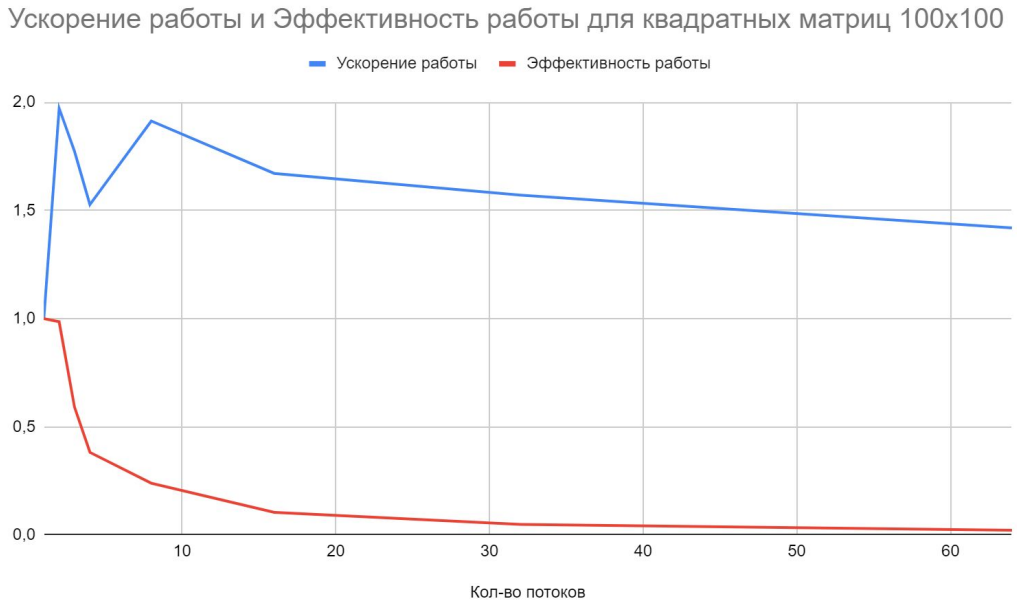
Я делал 4 замера времени работы 1, 2, 3, 4, 8, 32 и 64 потока(ов) и брал среднее значение. Затем я строил соответствующую таблицу.

Рассмотрим перемножение двух матриц размером 10x10:



Как видно по таблице, при перемножении маленьких матриц, эффективность потоков очень маленькая. И решение работает на одном потоке лучше, чем на нескольких. Это происходит из-за того, что процесс создания потоков и их поддержка не являются бесплатными, а полезное время работы этих потоков достаточно мало.

Рассмотрим перемножение двух матриц размером 100x100:



Здесь ситуация намного лучше. Так как здесь в 1000 раз больше вычислений, чем в прошлом примере, то доля полезных вычислений выше. Максимальное ускорение 1,97 достигается на двух потоках. Это связано с тем, что процессор, на котором производили тесты, является двухъядерным. При большем кол-ве процессов, начинается конкуренция за вычислительное время, а доля полезных вычислений уменьшается.

Основные файлы программы

main.c:

```
#include <time.h>
#include <stdio.h>
#include <complex.h>

#include "io.h"
#include "matrix.h"



Matrix *readMatrix() {
    const int STR_SIZE = 256;
    int n;
    int m;
    char str[STR_SIZE];
    get_line(str, STR_SIZE, '\n');

    sscanf(str, "%d %d", &n, &m);
    Matrix *matrix = m_init(n, m);

    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            double real;
            double img;

            get_line(str, STR_SIZE, ' ');
            sscanf(str, "%lf", &real);

            get_line(str, STR_SIZE, 'i');
            sscanf(str, "%lfi", &img);
            get_line(str, STR_SIZE, (j + 1 < m) ? ' ' : '\n');

            complex double *el = m_get(matrix, i, j);
            *el = real + img * I;
        }
    }
```

```
    return matrix;
}

void printMatrix(Matrix* matrix) {
    for(int i = 0; i < matrix->n; i++) {
        for(int j = 0; j < matrix->m; j++) {
            complex double z =  *m_get(matrix, i ,j);
            write_str(STDOUT_FILENO, "%.2lf %+.2lfi\t", creal(z), cimag(z));
        }
        write_str(STDOUT_FILENO, "\n");
    }
}

int main(int argc, char* argv[]) {
    if(argc != 2) {
        // USAGE;
        return 1;
    }
    int max_threads;
    if(sscanf(argv[1], "%d", &max_threads) != 1) {
        // ERROR INPUT
    }

    Matrix *matrixA = readMatrix();
    printf("Считывание закончено!\n");
    Matrix *matrixB = readMatrix();
    struct timespec t_start, t_end;
    clock_gettime (CLOCK_REALTIME, &t_start);

    Matrix *matrixC = m_product(max_threads, matrixA, matrixB);

    clock_gettime (CLOCK_REALTIME, &t_end);
    write_str(STDERR_FILENO, "Времени заняло: %lf\n",
        (double)(t_end.tv_nsec - t_start.tv_nsec) / 1000000000 + (double)(t_end.tv_sec -
t_start.tv_sec));
```

```
    printMatrix(matrixC);

    m_destroy(matrixC);
    m_destroy(matrixA);
    m_destroy(matrixB);
}
```

matrix.h:

```
#pragma once
#include <complex.h>

typedef struct {
    int n;
    int m;
    double complex *buf;
} Matrix;

Matrix* m_init(int n, int m);
void m_destroy(Matrix* matrix);

double complex *m_get(Matrix* matrix, int i, int j);

Matrix* m_product(int max_threads, Matrix* A, Matrix* B);
```

matrix.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

#include "matrix.h"
#include "io.h"

Matrix* m_init(int n, int m) {
   Matrix* matrix = malloc(sizeof(Matrix));
   if(matrix == NULL)
      return matrix;
```

```
    matrix->n = n;
    matrix->m = m;
    matrix->buf = malloc(sizeof(double complex)*n*m);

    if(matrix->buf == NULL) {
        free(matrix);
        return NULL;
    }

    {
        int size = n*m;
        for(int i = 0; i < size; i++) {
            matrix->buf[i] = 0;
        }
    }

    return matrix;
}

void m_destroy(Matrix* matrix) {
    if(matrix == NULL)
        return;

    free(matrix->buf);
    free(matrix);
}

double complex *m_get(Matrix* matrix, int i, int j) {
    if(matrix == NULL)
        return NULL;
    if(i < 0 || i >= matrix->n ||
      j < 0 || j >= matrix->m) {
        write_str(STDOUT_FILENO, " в ы х о д  з а  г р а н и ц ы  м а т р и ц ы ! %d %d", i, j);
        return NULL;
    }
```

```
    return &matrix->buf[i*matrix->m + j];
}

typedef struct {
    Matrix* A;
    Matrix* B;
    Matrix* C;

    int i_start;
    int i_end;
} m_product_arg;
void* m_product_thread(void* arg) {
    m_product_arg* prod_arg = (m_product_arg*)arg;
    for(int i = prod_arg->i_start; i < prod_arg->i_end; i++) {
        for(int j = 0; j < prod_arg->C->m; j++) {
            for(int k = 0; k < prod_arg->B->n; k++) {
                double complex z1 = prod_arg->A->buf[i * prod_arg->A->m + k];
                double complex z2 = prod_arg->B->buf[k * prod_arg->B->m + j];
                prod_arg->C->buf[i * prod_arg->C->m + j] += z1 * z2;
            }
        }
    }
    return NULL;
}

Matrix* m_product(int max_threads, Matrix* A, Matrix* B) {
    if(A->m != B->n)
        return NULL;
    if(A->n == 0 || B->m == 0)
        return m_init(A->n, B->m);
    Matrix* C = m_init(A->n, B->m);
    /*
    Берем кол-во строчек (A->n + max_threads - 1) / max_threads = Кол-во строчек на поток.
    threads = A->n / Кол-во строчек на поток.
```

```
    для каждого потока{
      Определить начало и конец.
      Запустить поток!
    }

    для каждого потока{
      дождаться завершения этого потока
    }
    */
    int row_pre_thread = (A->n + max_threads - 1) / max_threads;
    int threads = (A->n + row_pre_thread - 1) / row_pre_thread;
    m_product_arg thread_args[threads];
    pthread_t thread_id[threads];
    for(int thread = 0; thread < threads; thread++) {
      thread_args[thread].A = A;
      thread_args[thread].B = B;
      thread_args[thread].C = C;
      thread_args[thread].i_start = thread * row_pre_thread;

      thread_args[thread].i_end = (thread + 1) * row_pre_thread;
      if(thread_args[thread].i_end > A->n)
        thread_args[thread].i_end = A->n;

      if(pthread_create(&thread_id[thread], NULL, m_product_thread, (void*)&thread_args[thread]) != 0) {
        m_destroy(C);
        return NULL;
      }
    }
    for(int thread = 0; thread < threads; thread++) {
      pthread_join(thread_id[thread], NULL);
    }

    return C;
}
```

io.h:

```
#pragma once
#include "unistd.h"

// Считывает из потока ввода строку, максимальной длиной max_size,
// Строка должна заканчиваться знаком end.
// Возвращает длину считанной строки.
int get_line(char* str, int max_size, char end);

// Записывает в поток fd строку str. Возвращает кол-во записанных знаков
// Длина отформатированной строки не должна превышать 1024 символа!
// Возвращает длину записанной строки. Если -1, то возникла какая-то ошибка
int write_str(int fd, char* fomrat, ...);
```

io.c:

```
#include <stdio.h>
#include <stdarg.h>

#include "io.h"

int read_from_readed_buffer(char* out_str, int nbytes) {
    static const int BUF_MAX_SIZE = 4096;
    static char buf[4096]; // Здесь значение BUF_MAX_SIZE // Он думает, что BUF_MAX_SIZE не константное значение
    static int size = 0;
    static int pos = 0;

    int res_len = nbytes;
    while(nbytes > 0) {
        if(size == pos) {
            pos = 0;
            size = read(STDIN_FILENO, buf, BUF_MAX_SIZE);
            if(size < 0)
                return size;
            if(size == 0)
```

```
            break;
        }
        *out_str++ = buf[pos++];
        nbytes--;
    }

    return res_len - nbytes;
}

int get_line(char* str, int max_size, char end) {
    int str_len = 0;
    int max_size_str = max_size - 1;

    for(;str_len < max_size_str; str_len++) {
        if(read_from_readed_buffer(&str[str_len], 1) == 0)
            break;
        if(str[str_len] == end)
            break;
    }
    str[str_len] = '\0';

    return str_len;
}

int write_str(int fd, char* format, ...) {
    const int BUF_MAX_SIZE = 1024;
    char buf[BUF_MAX_SIZE];

    va_list args;
    va_start(args, format);
    int len = vsnprintf(buf, BUF_MAX_SIZE, format, args);
    va_end(args);

    int writed_bytes = 0;
    while(len > 0) {
```

```
        int writed = write(fd, buf + writed_bytes, len);
        if(write < 0)
            return -1;
        writed_bytes += writed;
        len -= writed;
    }

    return writed_bytes;
}
```

Пример работы

```
reterer@serv:~/OS/os_lab_3/src$ cat ../manual_test/01.t
2 2
1 +0i 0 +0i
0 +0i 1 +0i
2 2
1 +1i 2 +2i
3 +3i 4 +4i
reterer@serv:~/OS/os_lab_3/src$ make
gcc main.c io.c matrix.c -pthread -o lab3
reterer@serv:~/OS/os_lab_3/src$ ./lab3 2 <../manual_test/01.t
1.00 +1.00i     2.00 +2.00i
3.00 +3.00i     4.00 +4.00i
```

Вывод

Потоки, в отличие от процессов, являются более легкими структурами. Их быстрее создавать и переключать, они занимают меньше памяти. И кроме этого, они имеют общую память. Последнее упрощает взаимодействие между потоками, делает его более быстрым. Но в этом и кроется опасность: нужно очень внимательно разрабатывать многопоточные приложения, так как возможны взаимные блокировки и гонки данных. Нужно понимать, какие функции и структуры данных, операции, являются потоково-безопасными, а какие нет.

Для избежания этих проблем используются множество средств, таких как мьютексы, семафоры, каналы и т.д.

Потоки можно использовать для выполнения фоновых задач, например, автосохранения, или, например, для GUI.