Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

**Потоки в операционных системах**

| Студент | Гришин Алексей Юрьевич |
| --- | --- |
| Группа | М8О-208Б-21 |
| Вариант | 6 |
| Преподаватель | Соколов Андрей Алексеевич |
| Оценка | 5 |
| Дата | 03.10.2022 |
| Подпись |  |

Москва, 2022.

## **Постановка задачи**

### **Цель работы**

Целью является приобретение практических навыков в:

* Управление потоками в ОС
* Обеспечение синхронизации между потоками

### **Задание**

Составить программу на языке С, осуществляющую перемножение 2-ух матриц, содержащих комплексные числа, в многопоточном режиме. При обработке 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.

## **Общие сведения о программе**

### **Описание**

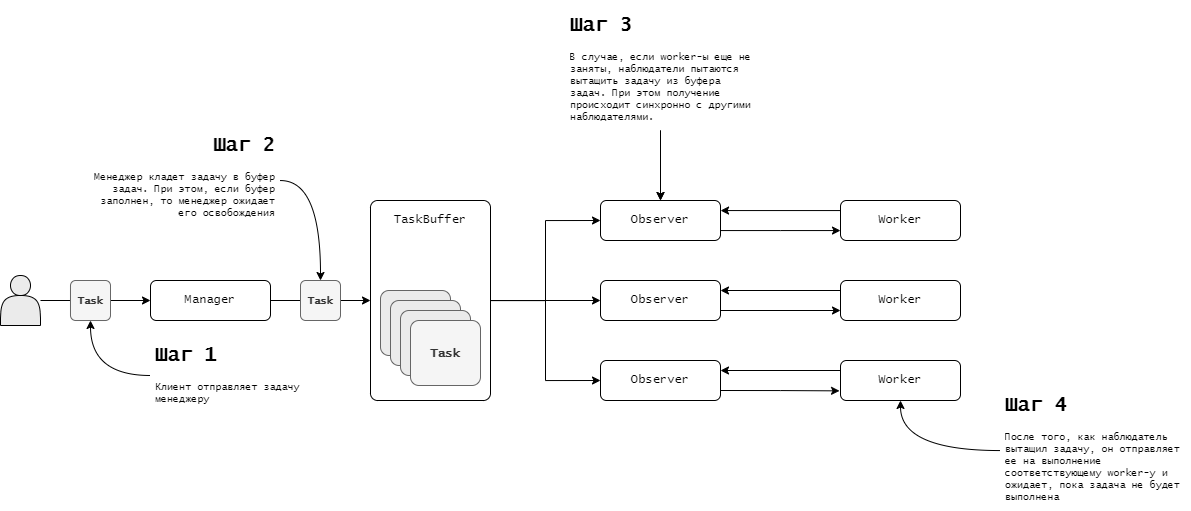
Программа состоит из одного главного файла main.c и вспомогательных компонентов: matrix, io\_matrix, manager, worker.

Модуль matrix вводит новый тип данных матрицы и добавляет набор методов, для работы с этим типом данных. Список методов приведен ниже:

1. create\_matrix – создает объект матрицы
2. delete\_matrix – удаляет объект матрицы, освобождая память, которую он до этого занимал
3. rows, cols – возвращает количество строк и столбцов матрицы соответственно
4. get\_cell, set\_cell – возвращает или устанавливает значение клетки матрицы по ее координатам

Модуль io\_matrix внедряет функционал, позволяющий считать матрицы с входного потока и вывести ее в отформатированном виде. Из описания данного модуля очевидно, что он содержит 2 метода: print\_matrix, scan\_matrix.

Основными модулями являются manager и worker. Хоть manager и worker и являются представлены в виде отдельных модулей, они являются единым компонентом. При реализации модулей я придерживался следующей архитектуры



Модуль worker представляет абстракцию потока, исполняющего задачу. Все, что необходимо для работы с такими объектами – наличие способа отправки задания и способа узнать о завершении выполнения задания. Следовательно, ключевую роль среди набора методов для работы с worker являются run\_task и wait\_worker.

Далее, согласно диаграмме выше, каждому worker-у соответствует наблюдатель. Задача наблюдателя заключается в обеспечении worker-а задачами. Делает он это путем получения задач из буфера задач. Буфер задач обеспечивает синхронные операции вставки и получения сообщений. По своей сути, данная задача очень похожа на задачу Consumer - Producer, поэтому основные идеи решения я взял оттуда. Также, стоит отметить, что при вставке задач в буфер необходимо учитывать момент, когда он заполнен. В этом случае мы должны заблокировать Producer-а до тех пор, пока в буфере не появится свободное место. Аналогичная ситуация и с получением задачи из буфера. Нам необходимо учитывать ситуацию, когда буфер пуст. В этом случае мы должны заблокировать Consumer-а до тех пор, пока в буфере не появится сообщение. Использование семафоров значительно облегчает реализацию этой задачи, так как их механика с использованием счетчика очень подходит задачи. В итоге, для буфера задач я использовал 2 семафора, значения которых обозначают количество свободных ячеек в буфере и количество свободных задач. Для обеспечения синхронизации получения и вставки сообщений я использовал блокировку mutex, так как она проста в использовании (все, что нам необходимо для работы с mutex – методы lock, unlock).

Задача менеджера заключается в структуризации и объединении отдельных частей всей системы в единое целое (своего рода реализация паттерна проектирования ”Фасад”). Внедрение менеджера позволяет сократить взаимодействие с системой до 2-х методов: do\_task и wait\_for\_task\_complete. Метод do\_task отправляет задачу на выполнение, а метод wait\_for\_task\_complete блокирует поток исполнения до тех пор, пока все задачи не будут выполнены. К тому же, при инициализации менеджера мы можем настроить размер буфера и количество worker-ов, что окажет большое удобство при дальнейшем анализе производительности программы.

### **Исходный код**

#### **io\_matrix/io\_matrix.h**

| #ifndef \_\_IO\_MATRIX\_H\_\_  #define \_\_IO\_MATRIX\_H\_\_  #include "../matrix/matrix.h"  #include <stdio.h>  typedef enum {  SCAN\_MATRIX\_SUCCESS = 0,  INVALID\_INPUT\_FORMAT = 1  } ScanMatrixExitCode;  void print\_matrix(FILE \*stream, PMatrix matrix);  ScanMatrixExitCode scan\_matrix(FILE \*stream, PMatrix \*matrix);  #endif |
| --- |

#### **io\_matrix/io\_matrix.c**

| #include "io\_matrix.h"  void print\_matrix(FILE \* stream, PMatrix matrix) {  for (int i = 0; i < rows(matrix); i++) {  for (int j = 0; j < cols(matrix); j++) {  MatrixElement el;  get\_cell(i, j, matrix, & el);  fprintf(stream, "%.2f + %.2fi\t", creal(el), cimag(el));  }  fprintf(stream, "\n");  }  }  ScanMatrixExitCode scan\_matrix(FILE \* stream, PMatrix \* matrix) {  int rows, cols;  float real, imag;  if (fscanf(stream, "%d %d", & rows, & cols) != 2)  return INVALID\_INPUT\_FORMAT;  create\_matrix(rows, cols, matrix);  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  if (fscanf(stream, "%f + %fi", & real, & imag) != 2)  return INVALID\_INPUT\_FORMAT;  set\_cell(i, j, CMPLX(real, imag), \* matrix);  }  }  return SCAN\_MATRIX\_SUCCESS;  } |
| --- |

#### 

#### **manager/observer/observer.c**

| #include "observer.h"  #include <pthread.h>  #include <stdlib.h>  struct \_\_Observer {  TaskBuffer observed\_buffer;  Worker worker;  pthread\_t observer\_thread\_id;  pthread\_mutex\_t end\_search\_lock;  };  void \_\_observer\_thread\_code(Observer observer) {  while (1) {  Task task = get\_task(observer -> observed\_buffer);  pthread\_mutex\_lock( & (observer -> end\_search\_lock));  run\_task(task, observer -> worker);  pthread\_mutex\_unlock( & (observer -> end\_search\_lock));  wait\_worker(observer -> worker);  }  }  void \* \_\_observer\_thread\_code\_wrapper(void \* data) {  Observer observer = (Observer) data;  \_\_observer\_thread\_code(observer);  pthread\_exit(0);  }  void create\_observer(TaskBuffer buffer, Worker worker, Observer \* observer) {  \* observer = (Observer) malloc(sizeof(struct \_\_Observer));  ( \* observer) -> observed\_buffer = buffer;  ( \* observer) -> worker = worker;  pthread\_mutex\_init( & ( \* observer) -> end\_search\_lock, NULL);  }  void start\_search(Observer observer) {  pthread\_create( &  observer -> observer\_thread\_id,  NULL,  \_\_observer\_thread\_code\_wrapper,  observer);  }  void end\_search(Observer observer) {  pthread\_mutex\_lock( & (observer -> end\_search\_lock));  pthread\_cancel(observer -> observer\_thread\_id);  pthread\_mutex\_unlock( & (observer -> end\_search\_lock));  }  void delete\_observer(Observer observer) {  pthread\_mutex\_destroy( & (observer -> end\_search\_lock));  free(observer);  } |
| --- |

#### **manager/observer/observer.h**

| #ifndef \_\_OBSERVER\_H\_\_  #define \_\_OBSERVER\_H\_\_  #include "../task\_buffer/task\_buffer.h"  #include "../../worker/worker.h"  typedef struct \_\_Observer \* Observer;  void create\_observer(  TaskBuffer buffer, Worker worker,  Observer \* observer);  void start\_search(Observer observer);  void end\_search(Observer observer);  void delete\_observer(Observer observer);  #endif |
| --- |

#### **manager/task\_buffer/task\_buffer.c**

| #include "task\_buffer.h"  #include <stdlib.h>  #include <pthread.h>  #include <semaphore.h>  struct \_\_TaskBuffer {  int free\_cell\_index;  Task \* buffer;  pthread\_mutex\_t edit\_buffer\_mutex;  sem\_t free\_cells;  sem\_t untaken\_tasks;  };  void create\_task\_buffer(int buffer\_size, TaskBuffer \* buffer) {  \* buffer = (TaskBuffer) malloc(sizeof(struct \_\_TaskBuffer));  ( \* buffer) -> free\_cell\_index = 0;  ( \* buffer) -> buffer = (Task \* ) calloc(buffer\_size, sizeof(Task));  pthread\_mutex\_init( & ( \* buffer) -> edit\_buffer\_mutex, NULL);  sem\_init( & ( \* buffer) -> free\_cells, 0, buffer\_size);  sem\_init( & ( \* buffer) -> untaken\_tasks, 0, 0);  }  void add\_task(Task task, TaskBuffer buffer) {  sem\_wait( & (buffer -> free\_cells));  pthread\_mutex\_lock( & (buffer -> edit\_buffer\_mutex));  buffer -> buffer[buffer -> free\_cell\_index] = task;  buffer -> free\_cell\_index++;  pthread\_mutex\_unlock( & (buffer -> edit\_buffer\_mutex));  sem\_post( & (buffer -> untaken\_tasks));  }  Task get\_task(TaskBuffer buffer) {  sem\_wait( & (buffer -> untaken\_tasks));  pthread\_mutex\_lock( & (buffer -> edit\_buffer\_mutex));  Task task = buffer -> buffer[buffer -> free\_cell\_index - 1];  buffer -> free\_cell\_index--;  pthread\_mutex\_unlock( & (buffer -> edit\_buffer\_mutex));  sem\_post( & (buffer -> free\_cells));  return task;  }  bool is\_empty(TaskBuffer buffer) {  pthread\_mutex\_lock( & (buffer -> edit\_buffer\_mutex));  bool result = buffer -> free\_cell\_index == 0;  pthread\_mutex\_unlock( & (buffer -> edit\_buffer\_mutex));  return result;  }  void delete\_task\_buffer(TaskBuffer buffer) {  pthread\_mutex\_destroy( & (buffer -> edit\_buffer\_mutex));  free(buffer -> buffer);  free(buffer);  } |
| --- |

#### 

#### **manager/task\_buffer/task\_buffer.h**

| #ifndef \_\_TASK\_BUFFER\_H\_\_  #define \_\_TASK\_BUFFER\_H\_\_  #include "../../task/task.h"  #include <stdbool.h>  typedef struct \_\_TaskBuffer \*TaskBuffer;  void create\_task\_buffer(int buffer\_size, TaskBuffer \*buffer);  void add\_task(Task task, TaskBuffer buffer);  Task get\_task(TaskBuffer buffer);  void delete\_task\_buffer(TaskBuffer buffer);  bool is\_empty(TaskBuffer buffer);  #endif |
| --- |

#### **manager/manager.c**

| #include "manager.h"  #include "observer/observer.h"  #include "task\_buffer/task\_buffer.h"  #include "../worker/worker.h"  #include <stdlib.h>  struct \_\_TaskManager {  Worker \* workers;  Observer \* observers;  TaskBuffer buffer;  int workers\_amount;  };  void \_\_init\_workers(int amount, Worker \*\* arr) {  \* arr = (Worker \* ) calloc(amount, sizeof(Worker));  for (int i = 0; i < amount; i++) create\_worker( & ( \* arr)[i]);  }  void \_\_init\_observers(  Worker \* workers,  int workers\_amount,  TaskBuffer buffer,  Observer \*\* arr)  {  \* arr = (Observer \* ) calloc(workers\_amount, sizeof(Observer));  for (int i = 0; i < workers\_amount; i++) {  create\_observer(buffer, workers[i], & ( \* arr)[i]);  start\_search(( \* arr)[i]);  }  }  void create\_task\_manager(  int workers\_amount,  int buffer\_size,  TaskManager \* manager)  {  \* manager = (TaskManager) malloc(sizeof(struct \_\_TaskManager));  ( \* manager) -> workers\_amount = workers\_amount;  create\_task\_buffer(  buffer\_size, &  ( \* manager) -> buffer);  \_\_init\_workers(  workers\_amount, &  ( \* manager) -> workers);  \_\_init\_observers(  ( \* manager) -> workers, workers\_amount,  ( \* manager) -> buffer, &  ( \* manager) -> observers);  }  void do\_task(Task task, TaskManager manager) {  add\_task(task, manager -> buffer);  }  void wait\_for\_task\_complete(TaskManager manager) {  while (!is\_empty(manager -> buffer));  for (int i = 0; i < manager -> workers\_amount; i++)  end\_search(manager -> observers[i]);  for (int i = 0; i < manager -> workers\_amount; i++)  if (is\_busy(manager -> workers[i]))  wait\_worker(manager -> workers[i]);  for (int i = 0; i < manager -> workers\_amount; i++)  start\_search(manager -> observers[i]);  }  void delete\_task\_manager(TaskManager manager) {  for (int i = 0; i < manager -> workers\_amount; i++) {  end\_search(manager -> observers[i]);  delete\_observer(manager -> observers[i]);  delete\_worker(manager -> workers[i]);  }  delete\_task\_buffer(manager -> buffer);  free(manager -> workers);  free(manager -> observers);  free(manager);  } |
| --- |

#### **manager/manager.h**

| #ifndef \_\_TASK\_MANAGER\_H\_\_  #define \_\_TASK\_MANAGER\_H\_\_  #include "../task/task.h"  typedef struct \_\_TaskManager \*TaskManager;  void create\_task\_manager(int workers\_amount, int buffer\_size, TaskManager \*manager);  void do\_task(Task task, TaskManager manager);  void wait\_for\_task\_complete(TaskManager manager);  void delete\_task\_manager(TaskManager manager);  #endif |
| --- |

#### **matrix/matrix.c**

| #include "matrix.h"  #include <stdlib.h>  #include <stdio.h>  struct \_\_MatrixSize {  int columns;  int rows;  };  struct \_\_Matrix {  struct \_\_MatrixSize size;  MatrixElement \* elements;  };  MatrixExitCode create\_matrix(int n, int m, PMatrix \* matrix) {  if (n <= 0 || m <= 0)  return INVALID\_MATRIX\_SIZE;  \* matrix = (PMatrix) malloc(sizeof(struct \_\_Matrix));  if ( \* matrix == NULL)  return CREATE\_MATRIX\_FAILED;  ( \* matrix) -> size.rows = n;  ( \* matrix) -> size.columns = m;  ( \* matrix) -> elements = (MatrixElement \* ) calloc(  n \* m, sizeof(MatrixElement));  if (( \* matrix) -> elements == NULL) {  free( \* matrix);  return CREATE\_MATRIX\_FAILED;  }  return MATRIX\_SUCCESS;  }  int rows(PMatrix matrix) {  return (matrix -> size).rows;  }  int cols(PMatrix matrix) {  return (matrix -> size).columns;  }  MatrixExitCode get\_cell(  int i, int j, PMatrix matrix,  MatrixElement \* out\_value)  {  if (i < 0 || i >= rows(matrix) || j < 0 || j >= cols(matrix))  return INVALID\_COORDINATES;  \* out\_value = matrix -> elements[i \* cols(matrix) + j];  return MATRIX\_SUCCESS;  }  MatrixExitCode set\_cell(int i, int j, MatrixElement value, PMatrix matrix) {  if (i < 0 || i >= rows(matrix) || j < 0 || j >= cols(matrix))  return INVALID\_COORDINATES;  matrix -> elements[i \* cols(matrix) + j] = value;  return MATRIX\_SUCCESS;  }  void delete\_matrix(PMatrix matrix) {  free(matrix -> elements);  free(matrix);  } |
| --- |

#### **matrix/matrix.h**

| #ifndef \_\_MATRIX\_H\_\_  #define \_\_MATRIX\_H\_\_  #include <complex.h>  typedef double complex MatrixElement;  typedef struct \_\_Matrix \* PMatrix;  typedef enum {  MATRIX\_SUCCESS = 0,  CREATE\_MATRIX\_FAILED = 1,  INVALID\_MATRIX\_SIZE = 2,  INVALID\_COORDINATES = 3  }  MatrixExitCode;  MatrixExitCode create\_matrix(int n, int m, PMatrix \* matrix);  int rows(PMatrix matrix);  int cols(PMatrix matrix);  MatrixExitCode get\_cell(int i, int j, PMatrix matrix, MatrixElement \* out\_value);  MatrixExitCode set\_cell(  int i, int j, MatrixElement value,  PMatrix matrix);  void delete\_matrix(PMatrix matrix);  #endif |
| --- |

#### **task/task.h**

| #ifndef \_\_TASK\_H\_\_  #define \_\_TASK\_H\_\_  #include "../matrix/matrix.h"  typedef struct {  PMatrix a, b;  int i, j;  PMatrix out\_matrix;  } Task;  #endif |
| --- |

#### **worker/worker.c**

| #include "worker.h"  #include "../matrix/matrix.h"  #include <pthread.h>  #include <stdbool.h>  #include <stdlib.h>  #include <stdio.h>  struct \_\_Worker {  pthread\_t id;  Task task;  bool is\_busy;  };  void \_\_worker\_code(PMatrix a, PMatrix b, int i, int j, PMatrix c) {  MatrixElement result = 0;  for (int p = 0; p < cols(a); p++) {  MatrixElement a\_element;  MatrixElement b\_element;  get\_cell(i, p, a, & a\_element);  get\_cell(p, j, b, & b\_element);  result += a\_element \* b\_element;  }  set\_cell(i, j, result, c);  }  void \* \_\_worker\_code\_wrapper(void \* data) {  Worker worker = (Worker) data;  \_\_worker\_code(  worker -> task.a, worker -> task.b,  worker -> task.i, worker -> task.j,  worker -> task.out\_matrix  );  worker -> is\_busy = false;  pthread\_exit(0);  }  void create\_worker(Worker \* worker) {  \* worker = (Worker) malloc(sizeof(struct \_\_Worker));  ( \* worker) -> is\_busy = false;  }  void run\_task(Task task, Worker worker) {  worker -> task = task;  worker -> is\_busy = true;  pthread\_create( & worker -> id, NULL, \_\_worker\_code\_wrapper, worker);  }  void wait\_worker(Worker worker) {  pthread\_join(worker -> id, NULL);  }  void delete\_worker(Worker worker) {  free(worker);  }  bool is\_busy(Worker worker) {  return worker -> is\_busy;  } |
| --- |

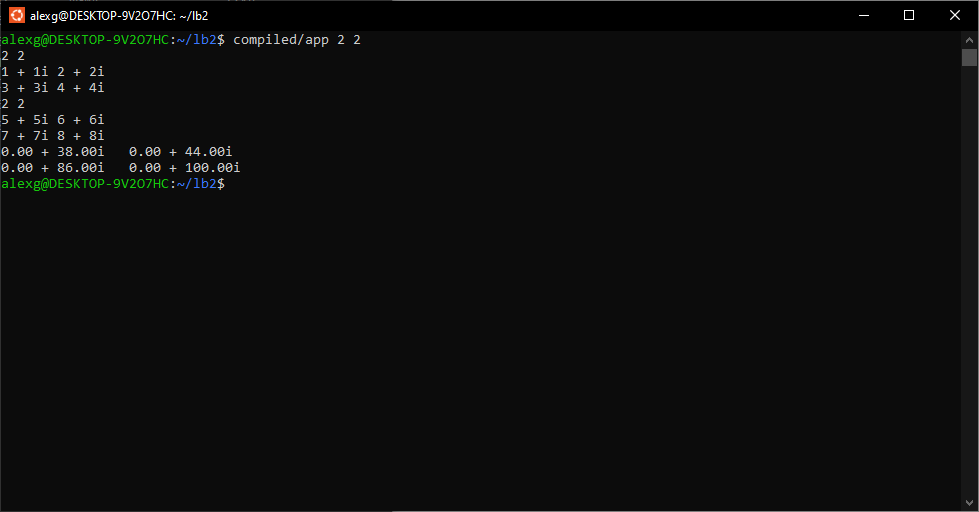
#### **worker/worker.h**

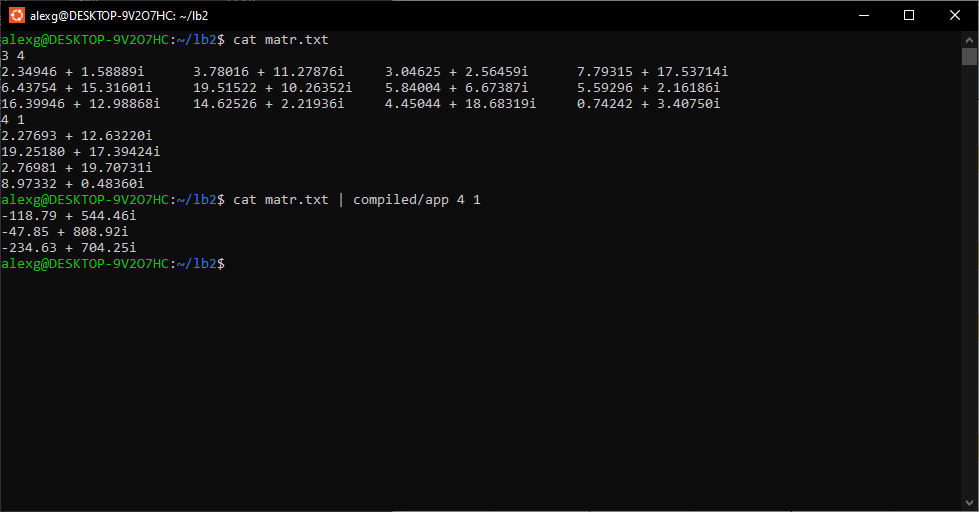
| #ifndef \_\_WORKER\_H\_\_  #define \_\_WORKER\_H\_\_  #include "../task/task.h"  #include <stdbool.h>  typedef struct \_\_Worker \*Worker;  void create\_worker(Worker \*worker);  void run\_task(Task task, Worker worker);  void wait\_worker(Worker worker);  void delete\_worker(Worker worker);  bool is\_busy(Worker worker);  #endif |
| --- |

#### **main.c**

| #include "matrix/matrix.h"  #include "io\_matrix/io\_matrix.h"  #include "manager/manager.h"  #include "task/task.h"  #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  int workers\_amount;  int buffer\_size;  void handle\_scan\_error(ScanMatrixExitCode code) {  if (code == INVALID\_INPUT\_FORMAT)  perror("[ScanMatrixError] Invalid input data format!\n");  exit(20 + code);  }  PMatrix multiply(PMatrix a, PMatrix b) {  if (cols(a) != rows(b)) {  perror("[MultiplyError] Invalid matrix sizes\n");  exit(31);  }  PMatrix c;  TaskManager manager;  create\_matrix(rows(a), cols(b), & c);  create\_task\_manager(workers\_amount, buffer\_size, & manager);  for (int i = 0; i < rows(c); i++) {  for (int j = 0; j < cols(c); j++) {  Task task = {  a,  b,  i,  j,  c  };  do\_task(task, manager);  }  }  wait\_for\_task\_complete(manager);  delete\_task\_manager(manager);  return c;  }  int main(int argc, char \* argv[]) {  PMatrix a, b, c;  int scan\_result;  workers\_amount = (argc >= 2) ? atoi(argv[1]) : 1;  buffer\_size = (argc >= 3) ? atoi(argv[2]) : 1;  scan\_result = scan\_matrix(stdin, & a);  if (scan\_result != SCAN\_MATRIX\_SUCCESS)  handle\_scan\_error(scan\_result);  scan\_result = scan\_matrix(stdin, & b);  if (scan\_result != SCAN\_MATRIX\_SUCCESS)  handle\_scan\_error(scan\_result);  c = multiply(a, b);  print\_matrix(stdout, c);  delete\_matrix(a);  delete\_matrix(b);  delete\_matrix(c);  return 0;  } |
| --- |

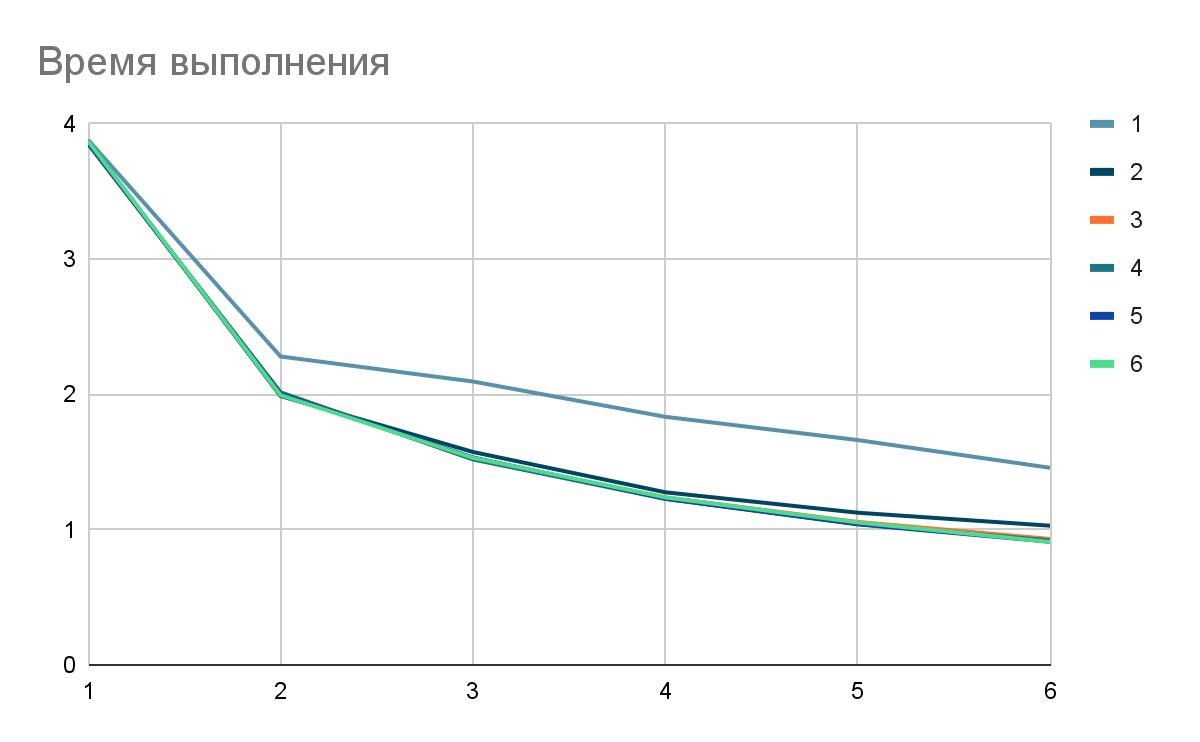
### **Пример работы**





## **Исследование программы**

Исследуем программы на производительность. По вышеперечисленным причинам при оценке эффективности системы я буду производить по двум параметрам: количеству потоков и размеру буфера сообщений. Ниже представлен график времени выполнения программы. Каждый график соответствует определенному размеру буфера, ось X показывает количество потоков, а ось Y – время исполнения. Анализ производился на двух матрицах размерами (в итоге, программа должна совершить 40 000 операций умножения комплексных чисел).



Как видим, увеличение количества потоков положительно влияет на время выполнения программы. Более того, мы можем заметить резкий спад времени при переходе от 1-го поточного исполнения к многопоточному. Также, мы можем заметить, что снижение времени исполнения происходит менее интенсивно после момента, когда размер буфера становится больше, чем количество потоков. Итого, мы можем сделать вывод, что самое эффективное соотношение потоков к размеру буфера будет 1 к 1.

## **Вывод**

В ходе выполнения данной лабораторной работы я ознакомился с реализацией многопоточных программ на языке программирования C, изучил основные подходы к написанию асинхронных программ, узнал о недостатках асинхронного подхода и способах синхронизации потоков через такие блокировку как семафор и mutex. Также, я изучил библиотеку pthread систем Unix, с помощью которой можно писать многопоточные программы.