МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №2
по курсу «Операционные системы»

Выполнил: Д. Н. Дружинин
Группа: М8О-207БВ-24
Преподаватель: Е. С. Миронов

Москва, 2025

Условие

Цель работы:

Приобретение практических навыков разработки многопоточных программ с использованием стандартных средств операционных систем (Windows API / POSIX Threads), изучение механизмов синхронизации потоков (семафоры, мьютексы), а также исследование производительности многопоточного алгоритма на основе свёртки матриц. Цель — понять, как изменяются время выполнения, ускорение и эффективность при разных уровнях параллелизма и размерах входных данных.

Задание:

- Считывает из файлов: прямоугольную матрицу вещественных чисел; ядро свёртки (матрица фильтра).
- Выполняет K-кратное применение фильтра к исходной матрице.
- Обрабатывает каждую строку матрицы в отдельном потоке, создавая один рабочий поток на строку.
- Ограничивает количество одновременно работающих потоков с помощью семафора.
- Значение ограничения передаётся ключом запуска программы.
- Использует средства порождения потоков ОС:
  - Windows: `CreateThread`, WinAPI-семафоры;
  - Linux: `pthread_create`, POSIX-семафоры.
- Записывает результат в выходной файл.
- Позволяет продемонстрировать количество реально создаваемых потоков средствами ОС.

Вариант: 13

Метод решения

Разрабатываемая программа реализует многопоточное выполнение свёртки прямоугольной матрицы вещественных чисел с использованием ядра фильтра. В основу решения положен классический подход к распараллеливанию по строкам: каждая строка исходной матрицы обрабатывается отдельным рабочим потоком, что позволяет эффективно загрузить процессор при больших размерах входных данных.

Обработка строки представляет собой последовательное вычисление значений результирующей матрицы путём наложения окна ядра на соответствующие элементы исходной матрицы. Для каждого положения окна выполняется вычисление суммы произведений элементов окна и ядра (дискретная двумерная свёртка).

Создание потоков осуществляется средствами операционной системы:
для Windows — функцией `CreateThread` и WinAPI-семафорами,
для Linux — функцией `pthread_create` и POSIX-семафорами. Семафор используется для ограничения максимального количества одновременно работающих потоков. Это позволяет контролировать уровень параллелизма и анализировать влияние числа потоков на производительность программы.

Алгоритм синхронизации следующий: перед запуском потока выполняется операция ожидания семафора, уменьшающая счётчик доступных ресурсов. После завершения обработки строки поток выполняет операцию освобождения семафора, тем самым разрешая запуск следующего потока. Такой подход предотвращает чрезмерное создание потоков и стабилизирует нагрузку на процессор.

Чтение исходных данных реализовано через обработку текстовых файлов:

- первый файл содержит размеры матрицы и её элементы;
- второй файл содержит размеры ядра и значения фильтра;
- третий файл используется для записи результата.

После загрузки данных в память программа последовательно создаёт рабочие потоки, каждый из которых получает индекс строки, которую необходимо обработать. Основной поток программы ожидает завершения всех рабочих потоков, после чего формирует итоговую матрицу и записывает её в выходной файл.

Выбранная архитектура позволяет:

- варьировать степень параллелизма за счёт изменения количества разрешённых потоков;
- сравнивать скорости исполнения при разных конфигурациях;
- измерять ускорение и эффективность многопоточной реализации;
- исследовать влияние размера ядра и входной матрицы на время работы.

Такой метод решения демонстрирует практическое использование потоков, семафоров и базовых примитивов синхронизации, а также позволяет оценить производительность многопоточного алгоритма на реальных данных.

Описание программы

Программа состоит из нескольких модулей, каждый из которых отвечает за свою часть функциональности. Структура проекта разделена на файлы для обеспечения удобства разработки, модульности и возможности кроссплатформенной сборки.

Структура проекта

- **main.cpp** — точка входа в программу. Содержит:
  - чтение входных файлов и загрузку матриц в память;
  - создание рабочих потоков;
  - управление семафором и ограничением числа одновременно работающих потоков;
  - ожидание завершения всех потоков;
  - запись результата в выходной файл.
- **matrix.h / matrix.cpp** — модуль работы с прямоугольной матрицей:

  - определение типа `Matrix`;
  - функции чтения матрицы из файла;
  - функции создания, копирования, получения размеров;
  - функция безопасного доступа к элементам.
- **convolution.h / convolution.cpp** — реализация двумерной свёртки:
  - функция выполнения свёртки над исходной матрицей;
  - функция обработки одной строки (выполняется в рабочем потоке);
  - вспомогательные инструменты для вычисления окна ядра.
- **os_wrapper.h** — единый интерфейс работы с потоками и семафорами.
- **os_wrapper_win.cpp** — реализация функций под Windows:
  - создание потоков: `CreateThread`;
  - ожидание потока: `WaitForSingleObject`;
  - семафоры: `CreateSemaphore`, `ReleaseSemaphore`.
- **os_wrapper_linux.cpp** — реализация функций под Linux:
  - создание потоков: `pthread_create`;
  - ожидание потоков: `pthread_join`;
  - POSIX-семафоры: `sem_init`, `sem_wait`, `sem_post`.

Основные типы данных

- `Matrix` — структура, содержащая:
  - количество строк и столбцов;
  - двумерный динамический массив значений;
  - методы для доступа к элементам.
- `ThreadArgs` — структура с аргументами, передаваемыми в поток:
  - указатель на исходную матрицу;
  - указатель на матрицу-результат;
  - указатель на ядро свёртки;
  - индекс строки, которую должен обработать поток;
  - размер ядра.

Основные функции

- `readMatrix(file)` — читает матрицу из текстового файла.
- `writeMatrix(file, M)` — записывает матрицу в файл.
- `convolveRow(args)` — вычисляет свёртку для одной строки (функция для потока).
- `convolveAll(...)` — выполняет создание потоков и контролирует семафор.

Используемые системные вызовы

В зависимости от операционной системы программа использует следующие системные вызовы и API-функции.

Windows (WinAPI)

- `CreateThread` — создание нового потока.
- `WaitForSingleObject` — ожидание завершения потока.
- `CreateSemaphore` — создание семафора.
- `ReleaseSemaphore` — освобождение семафора.

Linux (POSIX Threads)

- `pthread_create` — создание нового потока.
- `pthread_join` — ожидание завершения потока.
- `sem_init` — создание семафора.
- `sem_wait` — захват семафора.
- `sem_post` — освобождение семафора.

Эти вызовы обеспечивают параллельное выполнение обработки строк матрицы, а также синхронизацию потоков и ограничение уровня параллелизма.

Результаты

В результате выполнения лабораторной работы была разработана кроссплатформенная многопоточная программа, реализующая K-кратную свёртку матрицы с использованием ядра фильтра. Каждая строка матрицы обрабатывается отдельным потоком, а уровень параллелизма контролируется с помощью семафора, что позволяет ограничивать число одновременно работающих потоков и анализировать влияние этого параметра на производительность.

Ключевые особенности полученного решения:

- — кроссплатформенность: используются `pthread_create` и POSIX-семафоры под Linux, а также `CreateThread` и WinAPI-семафоры под Windows;
- — одна строка матрицы соответствует одному рабочему потоку, что обеспечивает естественное распараллеливание задачи свёртки;
- — семафор позволяет гибко задавать максимальное количество одновременно активных потоков;

- — применена модульная архитектура (матрица, свёртка, обёртка ОС, основной модуль);

- — предусмотрена возможность многократного применения фильтра (K итераций).

Для экспериментальной части были проведены замеры времени работы при размере матрицы 5000×5000 и ядре свёртки 21×21. Результаты времени выполнения T_p для различного числа потоков представлены на графике.

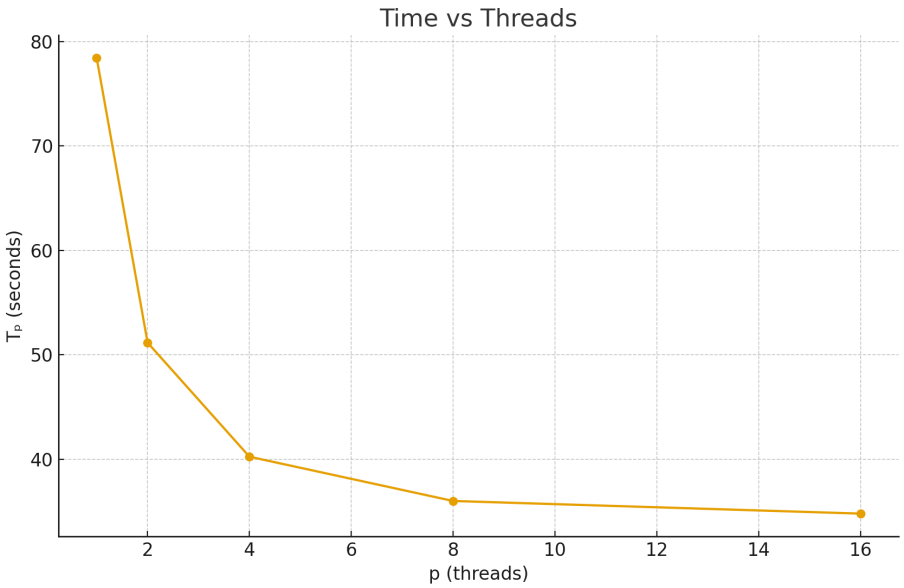

Рис. 1: Зависимость времени выполнения T_p от числа потоков p

На основе экспериментальных данных были вычислены ускорение S_p и эффективность X_p:

$$S_p = \frac{T_1}{T_p}, \qquad X_p = \frac{S_p}{p}$$

Полученные значения отражены на графиках:

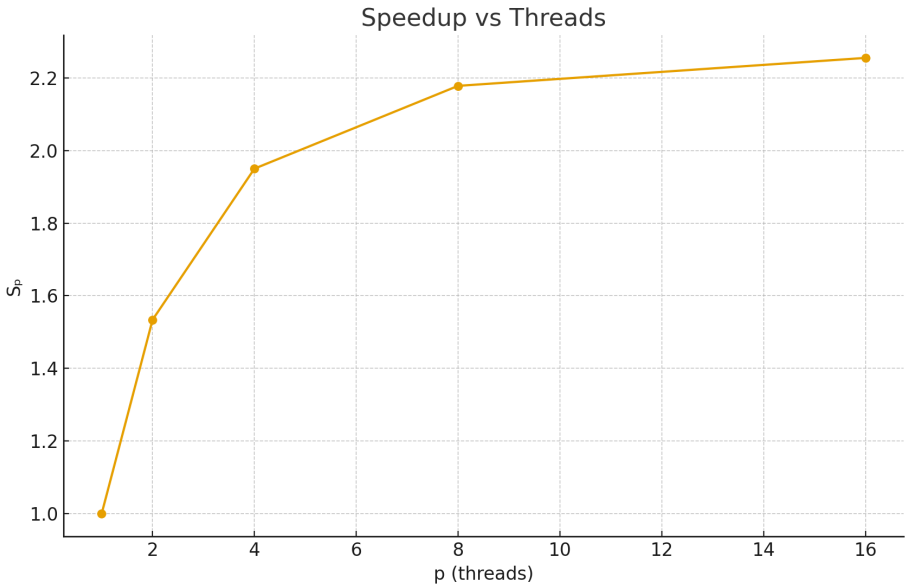

Рис. 2: Ускорение S_p

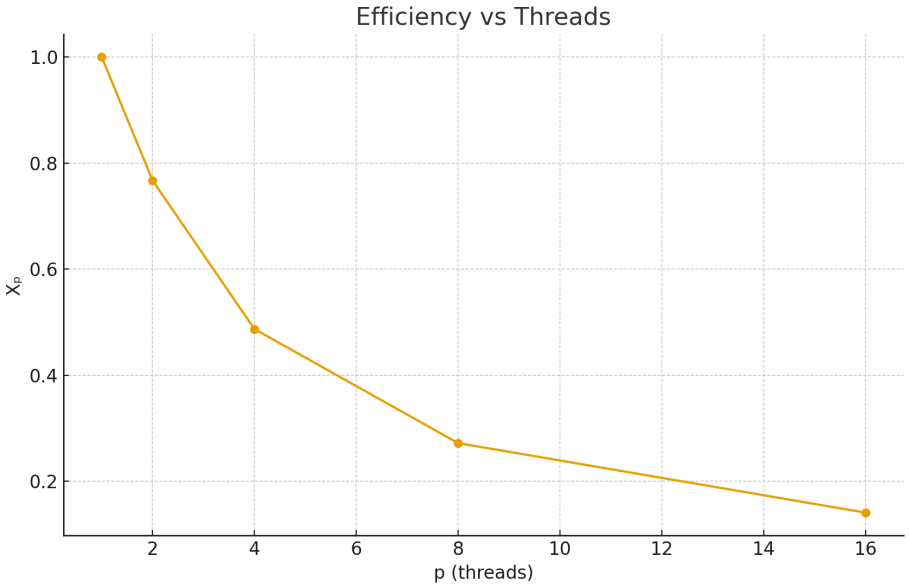

Рис. 3: Эффективность X_p

Дополнительно построен график величины $1/T_p$, демонстрирующий асимптотический характер роста производительности:

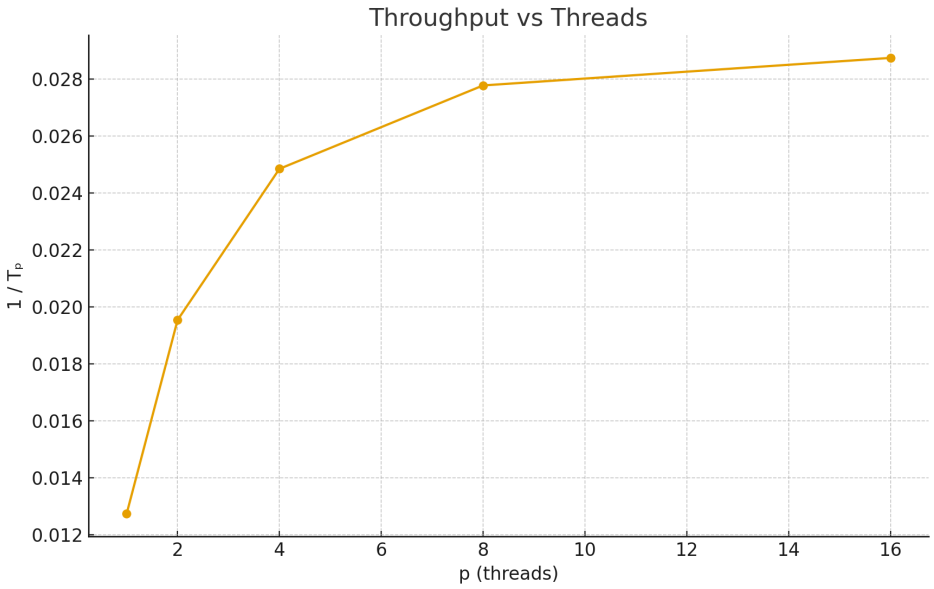

Рис. 4: График обратного времени $1/T_p$

Эксперимент показывает, что при малом числе потоков производительность растёт практически линейно. Однако начиная с $p \approx 8$ прирост замедляется. Это связано с ограниченной пропускной способностью памяти: операция двумерной свёртки является memory-bound, и конкуренция потоков за доступ к памяти снижает масштабируемость. Тем не менее ускорение остаётся значительным, что подтверждает эффективность параллельной обработки строк.

Выводы

В ходе выполнения лабораторной работы были получены практические навыки разработки многопоточных приложений и использования механизмов синхронизации (OS-семафоры, POSIX Threads, WinAPI Threads). Реализованная программа позволяет гибко управлять уровнем параллелизма и демонстрирует особенности поведения многопоточных алгоритмов на задаче двумерной свёртки.

Проведённые эксперименты показали следующее:

- При малом числе потоков ($p \leq 4$) ускорение растёт почти линейно.
- Начиная с $p \approx 8$ рост производительности замедляется из-за конкуренции потоков за доступ к оперативной памяти (операция свёртки является memory-bound).
- Максимальная эффективность достигается в области $p = 4$–8.
- Дальнейшее увеличение числа потоков приводит к росту накладных расходов из-за планировщика ОС и уменьшает эффективность X_p.
- Построенный график $1/T_p$ демонстрирует асимптотический характер: производительность растёт всё медленнее при увеличении числа потоков.

Разработанное решение подтвердило, что грамотное использование потоков и семафоров позволяет существенно ускорить вычисление свёртки, однако масштабируемость ограничена пропускной способностью памяти и архитектурой системы

Исходная программа

В данном разделе приведён полный исходный код программы, включая основные модули: функции свёртки, работу с матрицами, кроссплатформенные обёртки потоков и семафоров под Linux и Windows, а также главный модуль программы. Код приводится полностью и соответствует собранной версии.

Файл convolution.h

```
#pragma once
#include "matrix.h"
#include "os_wrapper.h"
#include <vector>

struct ConvTask {
  int row;
  Matrix* in;
  Matrix* out;
  const std::vector<double>* kernel;
  int W;
  os_semaphore* sem;
};

void* worker_conv(void* arg);

void apply_convolution(Matrix& m, const std::vector<double>& kernel,
int W, int K, int max_threads);
```

Файл matrix.h

```
#pragma once
#include <vector>
#include <string>

class Matrix {
  public:
  int rows, cols;
  std::vector<double> data;

  Matrix(int r = 0, int c = 0);
  double& at(int r, int c);
  const double& at(int r, int c) const;

  static Matrix readFromFile(const std::string& path);
  void writeToFile(const std::string& path) const;
};
```

Файл `os_wrapper.h`

```
#pragma once
#include <cstddef>

typedef void* (*os_thread_func)(void*);
```

```

struct os_thread;
struct os_semaphore;

os_thread* os_thread_create(os_thread_func fn, void* arg);
void os_thread_join(os_thread* t);
void os_thread_destroy(os_thread* t);

os_semaphore* os_semaphore_create(int initial);
void os_semaphore_wait(os_semaphore* s);
void os_semaphore_signal(os_semaphore* s);
void os_semaphore_destroy(os_semaphore* s);
```

Файл convolution.cpp

```
#include "convolution.h"

void* worker_conv(void* arg) {
  ConvTask* t = (ConvTask*)arg;

  int r = t->row;
  int rows = t->in->rows;
  int cols = t->in->cols;
  int W = t->W;
  int half = W / 2;

  for (int c = 0; c < cols; ++c) {
    double sum = 0.0;

    for (int i = 0; i < W; ++i) {
      int rr = r + i - half;

      for (int j = 0; j < W; ++j) {
        int cc = c + j - half;

        double pixel = 0.0;

        if (rr >= 0 && rr < rows && cc >= 0 && cc < cols) {
          pixel = t->in->at(rr, cc);
        }

        sum += pixel * (*t->kernel)[i * W + j];
      }
    }

    t->out->at(r, c) = sum;
  }

  os_semaphore_signal(t->sem);
  delete t;
  return nullptr;
}

void apply_convolution(Matrix& m,
const std::vector<double>& kernel,
int W,
```

```
  int K,
  int max_threads)
  {
    Matrix temp(m.rows, m.cols);
    Matrix* in = &m;
    Matrix* out = &temp;

    os_semaphore* sem = os_semaphore_create(max_threads);

    for (int iter = 0; iter < K; ++iter) {
      std::vector<os_thread*> threads;
      threads.reserve(m.rows);

      for (int r = 0; r < m.rows; ++r) {
        os_semaphore_wait(sem);

        ConvTask* t = new ConvTask{
          r, in, out, &kernel, W, sem
        };

        os_thread* th = os_thread_create(worker_conv, t);
        threads.push_back(th);
      }

      for (auto t : threads) {
        os_thread_join(t);
        os_thread_destroy(t);
      }

      Matrix* tmp = in;
      in = out;
      out = tmp;
    }

    if (in != &m)
    m = *in;

    os_semaphore_destroy(sem);
  }
```

Файл main.cpp

```
  #include "matrix.h"
  #include "convolution.h"
  #include <iostream>
  #include <fstream>
  #include <vector>

  int main(int argc, char** argv) {
    if (argc < 6) {
      std::cout << "Usage: conv <input_matrix> <output_matrix> "
      "<kernel_file> <K> <max_threads>\n";
      return 1;
    }

    std::string input_file = argv[1];
```

```
    std::string output_file = argv[2];
    std::string kernel_file = argv[3];

    int K = std::atoi(argv[4]);
    int max_thr = std::atoi(argv[5]);

    Matrix m = Matrix::readFromFile(input_file);

    std::ifstream kin(kernel_file);
    int W;
    kin >> W;

    std::vector<double> kernel(W * W);
    for (int i = 0; i < W * W; ++i)
    kin >> kernel[i];

    apply_convolution(m, kernel, W, K, max_thr);

    m.writeToFile(output_file);

    return 0;
}
```

Файл matrix.cpp

```
#include "matrix.h"
#include <fstream>
#include <stdexcept>

Matrix::Matrix(int r, int c) : rows(r), cols(c), data(r * c) {}

double& Matrix::at(int r, int c) {
    return data[r * cols + c];
}

const double& Matrix::at(int r, int c) const {
    return data[r * cols + c];
}

Matrix Matrix::readFromFile(const std::string& path) {
    std::ifstream in(path);
    if (!in)
    throw std::runtime_error("Cannot open input file");

    int r, c;
    in >> r >> c;

    Matrix m(r, c);
    for (int i = 0; i < r * c; ++i)
    in >> m.data[i];

    return m;
}

void Matrix::writeToFile(const std::string& path) const {
    std::ofstream out(path);
```

```
  out << rows << " " << cols << "\n";

  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j)
    out << at(i, j) << " ";
    out << "\n";
  }
}
```

Файл os_wrapper_linux.cpp

```
#ifndef _WIN32
#include "os_wrapper.h"
#include <pthread.h>
#include <semaphore.h>
#include <cstdlib>

struct os_thread {
  pthread_t t;
};

void* thread_start(void* pack) {
  os_thread_func fn = ((os_thread_func*)pack)[0];
  void* arg = ((void**)pack)[1];
  free(pack);
  return fn(arg);
}

os_thread* os_thread_create(os_thread_func fn, void* arg) {
  void** pack = (void**)malloc(sizeof(void*) * 2);
  pack[0] = (void*)fn;
  pack[1] = arg;

  os_thread* t = new os_thread;
  pthread_create(&t->t, NULL, thread_start, pack);
  return t;
}

void os_thread_join(os_thread* t) {
  pthread_join(t->t, NULL);
}

void os_thread_destroy(os_thread* t) {
  delete t;
}

struct os_semaphore {
  sem_t sem;
};

os_semaphore* os_semaphore_create(int initial) {
  os_semaphore* s = new os_semaphore;
  sem_init(&s->sem, 0, initial);
  return s;
}

```

```
46  void os_semaphore_wait(os_semaphore* s) {
47    sem_wait(&s->sem);
48  }
49
50  void os_semaphore_signal(os_semaphore* s) {
51    sem_post(&s->sem);
52  }
53
54  void os_semaphore_destroy(os_semaphore* s) {
55    sem_destroy(&s->sem);
56    delete s;
57  }
58
59  #endif
```

Файл os_wrapper_win.cpp

```
1   #ifdef _WIN32
2   #include "os_wrapper.h"
3   #include <windows.h>
4   #include <process.h>
5   #include <cstdlib>
6
7   struct os_thread {
8     HANDLE h;
9   };
10
11  unsigned __stdcall thread_start(void* pack) {
12    os_thread_func fn = ((os_thread_func*)pack)[0];
13    void* arg = ((void**)pack)[1];
14    free(pack);
15    fn(arg);
16    return 0;
17  }
18
19  os_thread* os_thread_create(os_thread_func fn, void* arg) {
20    void** pack = (void**)malloc(sizeof(void*) * 2);
21    pack[0] = (void*)fn;
22    pack[1] = arg;
23
24    os_thread* t = new os_thread;
25    t->h = (HANDLE)_beginthreadex(NULL, 0, thread_start, pack, 0, NULL);
26    return t;
27  }
28
29  void os_thread_join(os_thread* t) {
30    WaitForSingleObject(t->h, INFINITE);
31  }
32
33  void os_thread_destroy(os_thread* t) {
34    CloseHandle(t->h);
35    delete t;
36  }
37
38  struct os_semaphore {
39    HANDLE h;
```

```
};

os_semaphore* os_semaphore_create(int initial) {
  os_semaphore* s = new os_semaphore;
  s->h = CreateSemaphoreA(NULL, initial, 1000000, NULL);
  return s;
}

void os_semaphore_wait(os_semaphore* s) {
  WaitForSingleObject(s->h, INFINITE);
}

void os_semaphore_signal(os_semaphore* s) {
  ReleaseSemaphore(s->h, 1, NULL);
}

void os_semaphore_destroy(os_semaphore* s) {
  CloseHandle(s->h);
  delete s;
}

#endif
```

Файл CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(MultiConv CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

include_directories(include)

add_executable(conv
src/main.cpp
src/matrix.cpp
src/convolution.cpp
)

if(WIN32)
target_sources(conv PRIVATE src/os_wrapper_win.cpp)
target_link_libraries(conv PRIVATE ws2_32)
else()
target_sources(conv PRIVATE src/os_wrapper_linux.cpp)
target_link_libraries(conv PRIVATE pthread)
endif()

add_custom_target(run
COMMAND conv
\${CMAKE_SOURCE_DIR}/example/input.txt
\${CMAKE_SOURCE_DIR}/example/output.txt
\${CMAKE_SOURCE_DIR}/example/kernel.txt
1 4
DEPENDS conv
```

```
  WORKING_DIRECTORY \${CMAKE_BINARY_DIR}
  )
```

Фрагмент журнала системных вызовов (strace)

Ниже приведена реальная трассировка системных вызовов, полученная командой:

```
strace -f ./conv input.txt output.txt kernel.txt 1 1
```

```
execve("./conv", ["./conv", "input.txt", "output.txt",
"kernel.txt", "1", "1"], ...) = 0
brk(NULL) = 0x61e15bef3000
mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x76b68a5ce000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT
openat(AT_FDCWD, "input.txt", O_RDONLY) = 3
read(3, "3 3\n1 1 1\n1 1 1\n1 1 1\n", 8191) = 22
close(3) = 0

openat(AT_FDCWD, "kernel.txt", O_RDONLY) = 3
read(3, "3\n1 1 1\n1 1 1\n1 1 1\n", 8191) = 20
close(3) = 0

clone(...) = 169014
clone(...) = 169015
clone(...) = 169016

openat(AT_FDCWD, "output.txt",
O_WRONLY|O_CREAT|O_TRUNC, 0666) = 4
write(4, "3 3\n4 6 4 \n6 9 6 \n4 6 4 \n", 25) = 25
close(4) = 0

exit_group(0) = ?
+++ exited with 0 +++
```