МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Параллельные алгоритмы»
Тема: Основы работы с процессами и потоками

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург
2024

Цель работы.

Изучение основ работы с процессами и потоками.

Исследование зависимости между количеством потоков, размерами входных данных и параметрами целевой вычислительной системы.

Задание.

Выполнить умножение 2х матриц.

Входные матрицы вводятся из файла (или генерируются).

Результат записывается в файл.

1.1.

Выполнить задачу, разбив её на 3 процесса. Выбрать механизм обмена данными между процессами.

Процесс 1: заполняет данными входные матрицы (читает из файла или генерирует их некоторым образом).

Процесс 2: выполняет умножение

Процесс 3: выводит результат

1.2.1

Аналогично 1.1, используя потоки (std::threads)

1.2.2

Разбить умножение на P потоков (можно “наивным” способом по по строкам-столбцам).

Протестировать, сравнив результат вычислений с результатами из 1.2.1 / 1.1

Выполнение работы.

Для работы с матрицами как процессов, так и поток, был создан класс Матрицы. Данный класс хранит в себе значения строк и столбцов, а также саму матрицу, которая записана в виде одного вектора, это сделано с той целью, что при чтении или записи сокета байты должны идут подряд, без разрывов.

Созданы основные методы: создание матрицы, отправка в сокет и чтение из него, печать в файл, и переопределение операции умножения.

```
class Matrix {
public:
    int rows_;
    int cols_;
    std::vector<int> data_;

    Matrix(int rows = 0, int cols = 0);
    Matrix& operator=(const Matrix& other);
};

bool readRowsCols(int argc, char* argv[], int& rowsLeft, int& colsLeft, int& rowsRight, int& colsRight);
Matrix createMatrix(int rows, int cols);
bool sendMatrix(const Socket& socket, const Matrix& matrix);
bool readMatrix(const Socket& socket, Matrix& matrix);
void printMatrix(const Matrix& matrix, const std::string& filename);

Matrix operator*(const Matrix& leftMatrix, const Matrix& rightMatrix);

bool isNumber(const std::string& str, int& num);
```

Задача 1.1

В качестве механизма обмена данными между процессами были выбраны сокеты. Это программная конечная точка, обеспечивающая двунаправленную связь между процессами, независимо от их расположения в системе или даже за ее пределами.

/tmp/read_socket и */tmp/write_socket* использовались для чтения и записи соответственно.

Было создано два класса *Socket* и *ServerSocket*. Первый выступал в роли клиента, второй в роли сервера.

Основные используемые методы в классе *Socket*:

socket() – создаёт конечную точку соединения и возвращает файловый дескриптор, указывающий на эту точку. Внутри метода указывался домен соединения (Локальное соединение), семантика соединения (Обеспечивает работу последовательного двустороннего канала).

connect() – устанавливает соединение с сокетом, заданным файловый дескриптором *sockfd*, ссылающимся на адрес *addr*.

Также использовалась структура *struct sockaddr_un*.

```
struct sockaddr_un {
    sa_family_t     sun_family;     /* Address family */
    char            sun_path[];     /* Socket pathname */
};
```

```
class Socket {
protected:
int fileDescriptor_;

public:
    Socket(int fileDescriptor = -1);
    Socket(const std::string& fileName);

    Socket(Socket&& other);

    virtual ~Socket();

    bool isValid() const;
    bool writeData(const int* data, int dataSize) const;
    bool readData(int* data, int dataSize) const;
};
```

Основные используемые методы в классе *ServerSocket*:

bind() назначает адрес, заданный в *addr*, сокету, указываемому дескриптором файла *sockfd*.

listen() помечает сокет, указанный в *sockfd* как пассивный, то есть как сокет, который будет использоваться для приёма запросов входящих соединений с помощью *accept()*.

Системный вызов *accept()* используется с сокетами, ориентированными на установление соединения.

```
class ServerSocket : public Socket {
private:
    struct sockaddr_un address_;

public:
    ServerSocket(const std::string& fileName);
    ~ServerSocket();

    Socket acceptConnection();
};
```

Также были реализованы методы для чтения и записи данных в сокет на основе знаний файлового дескриптора и занимаемого количества байт данными.

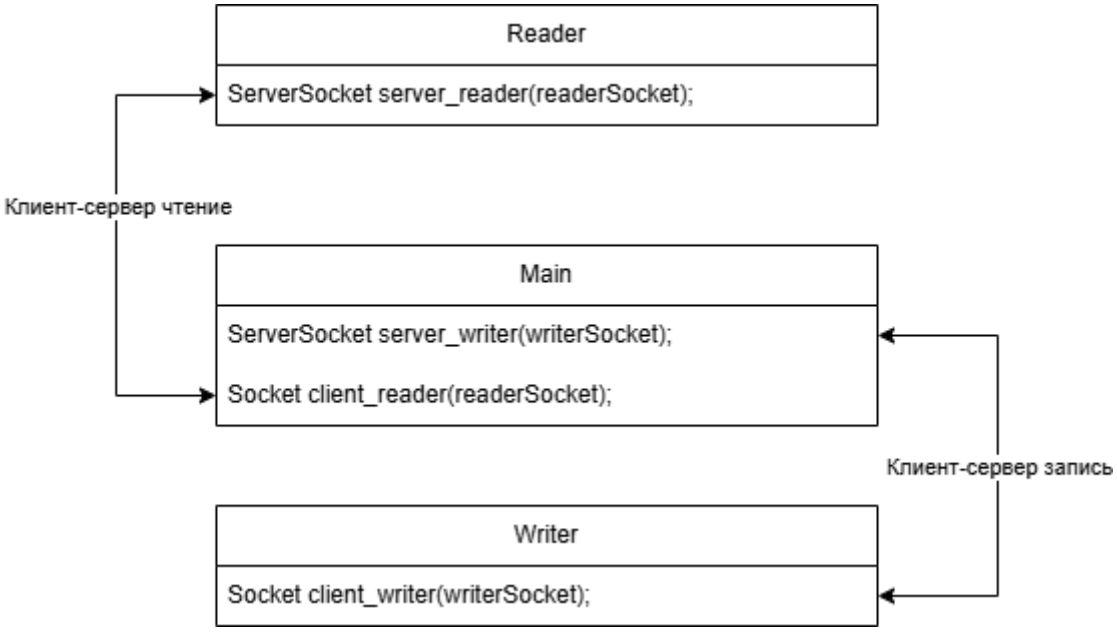
Для перемножения матриц было создано 3 процесса:

Reader – чтение матриц (чтение строк и столбцов и создание);

Main – перемножение матриц;

Writer – запись результата умножения матриц в файл.

Общение между процессами можно представить следующим образом:



Задача 1.2.1

В рамках выполнения данной задачи создавалось аналогичных с процессами три потока. Для ожидания завершения выполнения поток использовался метод *join()*.

```
1. std::thread readerThread(createMatrices, leftRows, leftCols, rightRows, right
   Cols);
2. readerThread.join();
3.
4. std::thread mainThread(multiplicationMatrices);
5. mainThread.join();
6.
7. std::thread writerThread(outputMatrix);
8. writerThread.join();
```

Задача 1.2.2

Задача по созданию P потоков аналогична предыдущей кроме создания основного потока. Для того, чтобы процесс умножения происходил по P потокам использовался следующий метод:

```
1. void multiplicationMatrices(int threadCount) {
2.     resultMatrix = Matrix(leftMatrix.rows_, rightMatrix.cols_);
3.     std::vector<std::thread> threads;
4.     threads.reserve(threadCount);
5.
6.     int rowsPerThread, remainingRows;
7.
8.     if (resultMatrix.rows_ < threadCount) {
9.         rowsPerThread = 1;
10.        remainingRows = 0;
11.    } else {
12.        rowsPerThread = resultMatrix.rows_ / threadCount;
13.        remainingRows = resultMatrix.rows_ % threadCount;
14.    }
15.
16.    int startRow = 0;
17.    for (int i = 0; i < (resultMatrix.rows_ < threadCount ? resultMatrix.rows
   _ : threadCount); i++) {
18.        int endRow = startRow + rowsPerThread + (i < remainingRows ? 1 : 0);
19.        threads.emplace_back(parallelMultiplication, startRow, endRow);
20.        startRow = endRow;
21.    }
22.
23.    for (auto& thread : threads) {
24.        thread.join();
25.    }
26.}
```

Такой метод предполагает равномерное распределение строк для умножения между всеми потоками.

Замеры времени выполнения программы

Чтобы произвести замеры и сравнить результат работы воспользуемся утилитой *time*, а для систематизированного запуска создадим скрипт *bash*.

```
1. run_type="${1}"
2.
3. case "${run_type}" in
4.     process)
5.         time {
6.             for i in $(seq 100); do
7.                 ./process/start.sh
8.             done
9.         }
```

```
    ;;
    3thread)
        time {
            for i in $(seq 100); do
                ./object/3_thread 150 150 150 150
            done
        }
    ;;
    pthread)
        time {
            for i in $(seq 100); do
                ./object/p_thread 30 30 30 30 2
            done
        }
    ;;
esac
```

Конкретнее были измерены следующие показатели:

- Реальное время, прошедшее с момента начала до конца выполнения.
- Время, проведённое в пользовательском режиме (то есть время, затраченное на выполнение программного кода).
- Время, проведённое в системном режиме (то есть время, затраченное на системные вызовы).

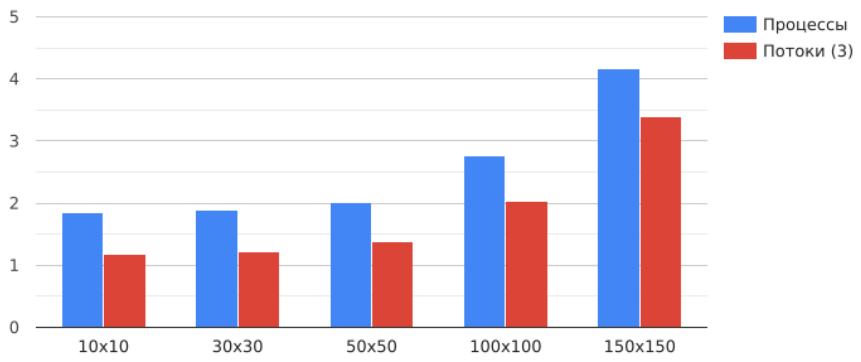
Процессы.

	Real, s	User, s	Sys, s
10x10	1.844	0.604	0.133
30x30	1.888	0.650	0.104
50x50	2.022	0.689	0.156
100x100	2.759	1.173	0.216
150x150	4.176	2.404	0.202

3 Потока.

	Real, s	User, s	Sys, s
10x10	1.179	0.175	0.048
30x30	1.227	0.205	0.037
50x50	1.372	0.228	0.050
100x100	2.039	0.362	0.083
150x150	3.399	1.321	0.172

Сравнение 3-х процессов и потоков.

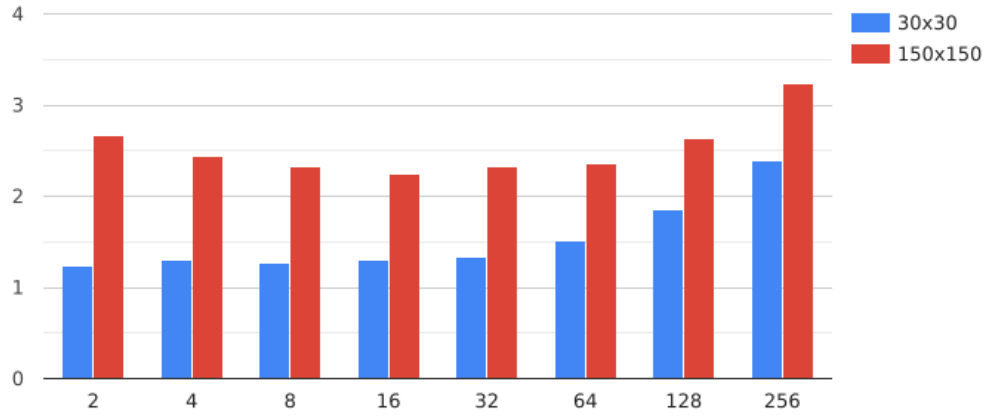


Глядя на диаграмму, можно сделать вывод, что потоки немного, но быстрее процессов. Это связано с тем, что они легковесные и используют общее адресное пространство, что снижает накладные расходы на создание и взаимодействие. Переключение контекста между потоками также быстрее, так как они не требуют изменения адресного пространства, как в случае с процессами.

n потоков

	Потоки, шт	Real, s	User, s	Sys, s
30x30	2	1.231	0.210	0.058
150x150	2	2.670	0.857	0.133
30x30	4	1.293	0.216	0.077
150x150	4	2.433	0.754	0.142
30x30	8	1.271	0.255	0.090
150x150	8	2.330	0.770	0.124
30x30	16	1.305	0.257	0.116
150x150	16	2.249	0.926	0.135
30x30	32	1.335	0.267	0.155
150x150	32	2.321	1.074	0.307
30x30	64	1.509	0.319	0.218
150x150	64	2.354	0.948	0.350

30x30	128	1.846	0.304	0.496
150x150	128	2.640	0.993	0.510
30x30	256	2.396	0.083	1.195
150x150	256	3.236	0.786	1.131



Глядя на диаграмму, можно сделать вывод, что выполнение многопоточных вычислений зависит как от размера матрицы, так и от количества потоков.

Для небольших матриц параллелизм может не дать значительного прироста производительности, так как время выполнения операции невелико. Увеличение размера матрицы приводит к большему количеству операций, что делает параллелизм более эффективным.

Для небольших матриц может быть достаточно одного потока, чтобы выполнить операцию быстро. Но при увеличении размера матриц использование нескольких потоков может помочь распределить нагрузку.

Увеличение числа потоков может улучшить производительность на больших матрицах, так как каждая часть задачи может выполняться параллельно. Однако важно учитывать, что слишком большое количество потоков может привести к уменьшению производительности из-за накладных расходов на управление потоками и конкуренции за ресурсы.

Также можно заметить, что наименьшее время затрачивается, когда количество потоков равно количеству процессоров системы.

Это связано с тем, что в случае меньшего количества потока, часть процессоров простаивается, а при большем процессорам приходится тратить дополнительное время на переключение между потоками.

Выводы.

В ходе выполнения лабораторной работы были изучены основы работы с потоками и процессами.

Была исследована зависимость между количеством потоков, размерами входных данных и параметрами целевой вычислительной системой. Сделан вывод, что чем ближе число потоков равно количеству процессоров, тем быстрее выполняются вычисления.