Отчет
о выполнении лабораторной работы №2
по дисциплине "Операционные системы"

Работу выполнил
Студент гр. ИТ-1-2024,
2 курс
Жуков М.А

1. Постановка задачи

Лабораторная работа №2 состоит из двух частей:

- реализовать умножение двух матриц N x N через n потоков
- реализовать умножение двух матриц N x N через n процессов

Требования

- на оценку "удовлетворительно" нужно сделать только задание с потоками, на оценку "хорошо" нужно сделать задание с процессами, на "отлично" сделать обе части (итоговая оценка, как и всегда, зависит от ответов на вопросы)
- при запуске программы пользователь задает размер матрицы и количество потоков (процессов), желательно делать это через аргументы командной строки
- нужно обязательно сравнить время работы в зависимости от количества потоков/процессов и построить сравнительные графики. Объяснить результат:
    i. Почему при малом числе потоков ускорение близко к линейному.
    ii. Почему при большом числе потоков/процессов производительность падает (переключение контекста, кэш, накладные расходы IPC).
    iii. Если делайте обе части, то почему процессы работают быстрее/медленнее/одинаково по сравнению с потоками (но это разумно сравнивать, только если обе части реализованы на одном языке программирования).
    iv. Какое количество процессов/потоков эффективнее всего.
- рекомендуется выполнить не менее 3 замеров для каждого числа потоков/процессов и усреднить результат. А еще лучше сделать около 10 замеров и удалить самое больше и самое маленькое значение и усреднить оставшийся результат.
- если делайте обе части (на "отлично"), то также сравнить время работы между потоками и процессами. Объяснить результат
- числа в матрице генерируются случайным образом, время генерации матрицы не входит в общее время замера: замерять нужно только время перемножения матриц
- обязательно где-то должна быть итоговая матрица - результат перемножения (были кадры, которые перемножали матрицы, но не "собирали" получающийся результат)
- на дополнительные полбалла (либо к этой лабе, либо к предыдущей, если у вас там меньше 5.0, либо к будущей) написать unit тесты, которые

проверяют результат перемножения матриц (тесты - это база в любом мало-мальски серьезном проекте)

- не забывайте о том, что в Питоне потоки не могут работать одновременно из-за GIL, поэтому на Питоне задание на потоки делать не надо, используйте другие языки (ну или изучите питоновские интерпретаторы, отключающие GIL)
- для хранения матриц используйте стандартные структуры данных выбранного языка. Не нужно использовать специализированные библиотеки и прочее, потому что цель лабораторной - вручную поработать с процессами и потоками
- до защиты подготовить графики, чтобы было, что обсуждать, без графиков даже не приходите сдавать лабу
- если вы делайте часть с процессами, то продумайте механизм их "общения" (помним, что каждый процесс имеет свое адресное пространство, в отличие от потоков, которые живут в одном адресном пространстве процесса)
- подбирайте такие размеры матриц, чтобы время перемножения матриц было достаточно большим (хотя бы несколько секунд). Это позволит получить более точные и интересные графики времени работы программы в зависимости от количества процессов/потоков
- код запускаем на хостовой ОС, не на виртуалке
- имейте в виду, что я буду в обязательном порядке спрашивать на понимание кода. Если понимания не будет - оценка неуд.

2. Процесс решения

2.1 Реализация с потоками

Программа использует библиотеку POSIX Threads (pthread) для создания и управления потоками. Основная идея заключается в разделении работы по умножению матриц между несколькими потоками, где каждый поток обрабатывает определенный набор строк результирующей матрицы.

Шаг 1: Инициализация

- Считываются параметры из командной строки или через консоль (размер матрицы и количество потоков)
- Выделяется память для трех матриц размером matrixSize
- Матрицы A и B заполняются случайными числами от 1 до 10 с помощью функции rand()

Шаг 2: Распределение работы между потоками
Строки матрицы распределяются равномерно:

- Рассчитывается количество строк матрицы для каждого потока rowsPerThread и количество дополнительных строк extraRows
- Каждый поток получает rowsPerThread строк
- Первые extraRows потоков получают по одной дополнительной строке

Шаг 3: Создание потоков
Каждый поток создается с помощью pthread_create() и получает указатель на свой набор начальной и конечной строк.

Шаг 4: Функция умножения выполняется каждым потоком
Каждый поток:

- Получает свой диапазон строк
- Для каждой строки вычисляет все элементы результирующей матрицы
- Использует стандартную формулу умножения матриц

Шаг 5: Ожидание завершения потоков
Главный поток ждет завершения всех рабочих потоков с помощью pthread_join().

2.2 Реализация с процессами

Программа использует системные вызовы POSIX для создания процессов fork() и разделяемую память mmap() для обмена данными между процессами. Каждый дочерний процесс обрабатывает свою часть результирующей матрицы.

Шаг 1: Инициализация и выделение разделяемой памяти
Используются одномерные массивы для упрощения работы с разделяемой памятью. Элемент матрицы [i][j] хранится по индексу i * matrixSize + j.

Шаг 2: Заполнение матриц
Матрицы A и B заполняются случайными числами, матрица C инициализируется нулями.

Шаг 3: Создание процессов
Системный вызов fork():

- Создает копию текущего процесса
- Дочерний процесс выполняет вычисления и завершается через exit(0)
- Родительский процесс продолжает создавать новые процессы

Шаг 4: Вычисления в дочернем процессе
Каждый дочерний процесс:

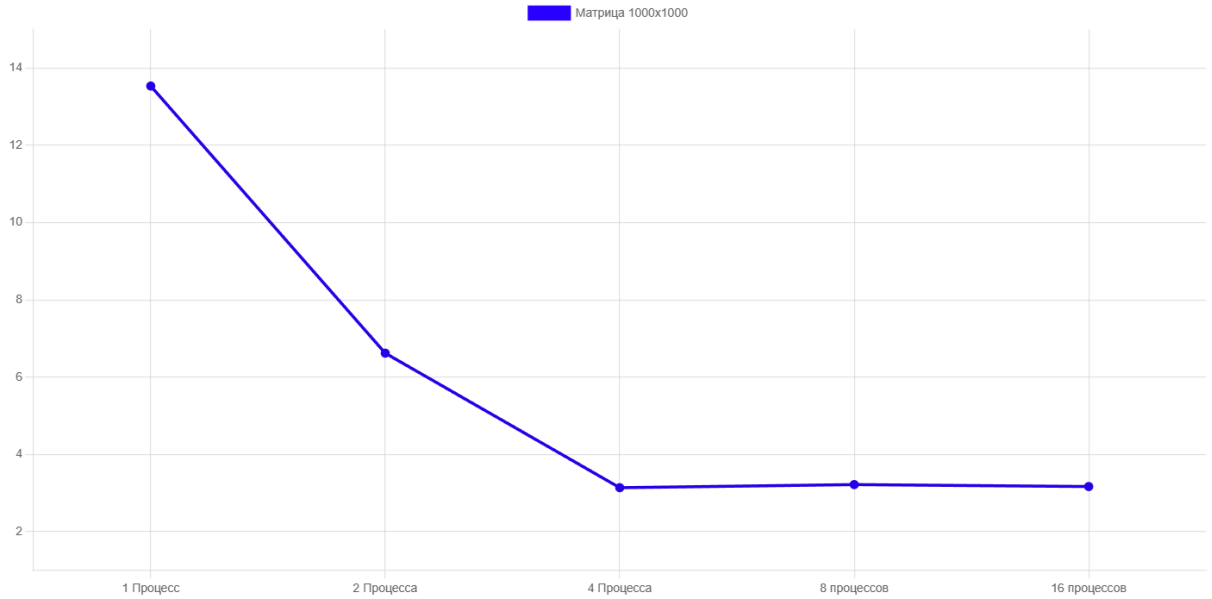
- Получает свой диапазон строк через переменные startRow и endRow
- Вычисляет элементы результирующей матрицы для своих строк
- Записывает результаты в разделяемую память
- Завершается через exit(0)

Шаг 5: Ожидание завершения процессов
Родительский процесс ждет завершения всех дочерних процессов с помощью wait().

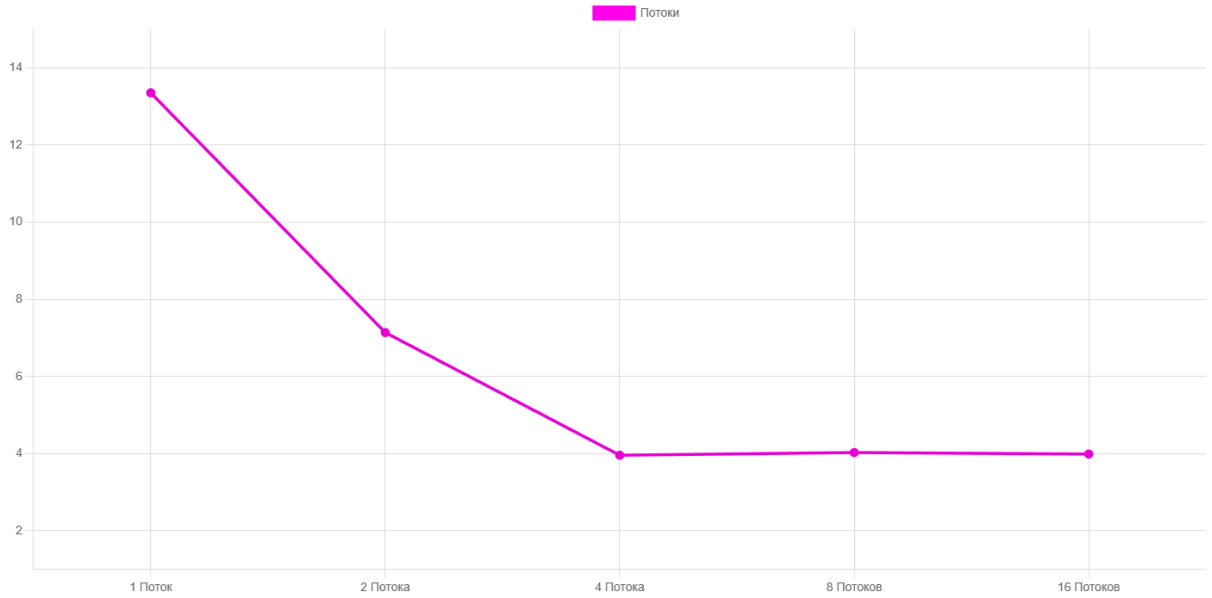
3. Полученные графики

Все точки на графиках являются усредненным значением 10 запусков программы с заданными параметрами, исключая самый малый и самый большой результаты.

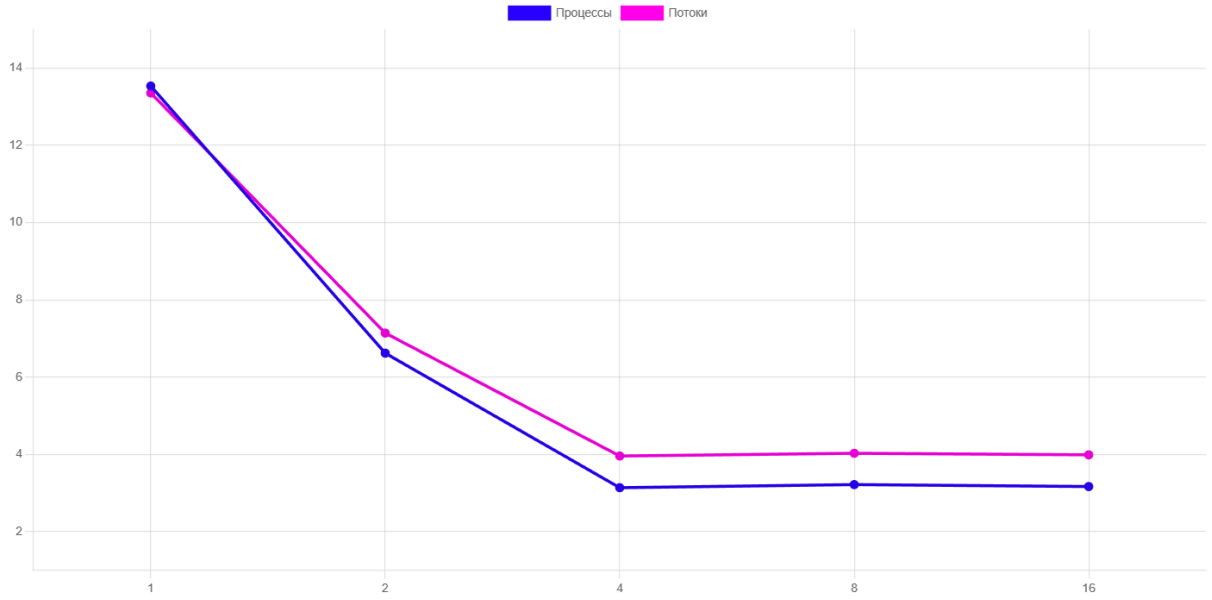
3.1 Тестирование процессорной версии на матрице 1000x1000



3.2 Тестирование потоковой версии на матрице 1000x1000



3.3 Общий график тестирований на матрице 1000x1000



4. Анализ результатов

Результаты

Количество	Процессы	Потоки	Ускорение (процессы)	Ускорение (потоки)
1	13.53	13.35	1.00	1.00
2	6.62	7.14	2.04	1.87
4	3.14	3.96	4.31	3.37
8	3.22	4.03	4.20	3.31
16	3.17	3.99	4.27	3.35

Эффективность

Количество	Эффективность (процессы)	Эффективность (потоки)
1	100%	100%
2	102%	93.5%
4	107.8%	84.3%
8	52.5%	41.4%
16	26.7%	20.9%

4.1 Почему при малом числе потоков ускорение близко к линейному.

При использовании 2-4 потоков/процессов наблюдается ускорение, близкое к идеальному (линейному). Это объясняется следующими факторами:

1. Оптимальное количество физических ядер

Количество потоков (2-4) не превышает количество физических ядер процессора:

- Каждый поток/процесс может выполняться на отдельном ядре
- Отсутствует конкуренция за процессорное время
- Минимальное переключение контекста между потоками

2. Хорошее разделение данных в кэше

При малом числе потоков:

- Каждое ядро работает с достаточно большим блоком данных
- Данные эффективно кэшируются в L1/L2 кэше ядра
- Минимальное количество промахов кэша
- Нет конфликтов за кэш между разными ядрами

3. Минимальные расходы на управление потоками/процессами

Накладные расходы на управление потоками/процессами:

- Создание 2-4 потоков занимает микросекунды
- Синхронизация (join/wait) происходит всего 2-4 раза
- Время на управление < времени вычислений

4.2 Почему при большом числе потоков производительность падает

При увеличении количества потоков/процессов до 8-16 наблюдается уменьшение производительности - время выполнения перестает уменьшаться и даже может увеличиваться из за:

1. Переключение контекста

Когда количество потоков/процессов превышает количество физических ядер (обычно 4-8), операционная система вынуждена постоянно переключаться между ними.
Накладные расходы:

- Одно переключение контекста: ~1-10 микросекунд
- При 16 потоках на 4 ядрах: тысячи переключений в секунду
- Суммарные потери: до 10-20% процессорного времени

Почему процессы страдают меньше: Это может связано с тем, что процессы имеют раздельные адресные пространства, и планировщик Linux может более эффективно распределять их по ядрам.

2. Конфликты кэша
Так как ядра процессора имеют общий L3 кэш, могут произойти конфликты между ядрами:

- Ложное разделение (Разные потоки работают с одной кэш линией)
- Вытеснение данных из кэша
- Cache Thrashing (Вытеснение кэша в медленную RAM память)

3. Накладные расходы IPC (для процессов)
В реализации процессы используют разделяемую память, но имеют скрытые накладные расходы:

fork() создает:

- Копию таблицы страниц процесса
- Новый дескриптор процесса в ОС
- Новое адресное пространство (даже если память разделяемая)

16 процессов = 16 вызовов fork():

- Каждый fork() занимает ~100-500 мкс
- 16 × 500 мкс = 8000 мкс = 8 мс
- Плюс затраты на управление процессами

4.3 Почему процессы работают быстрее потоков

Процессы работают на 7-20% быстрее потоков, благодаря:

- Copy-on-Write: fork() копирует только измененные страницы памяти, а не всю матрицу сразу
- Изоляция кэша: Каждый процесс использует свой L1/L2 кэш CPU, нет конфликтов.
- Нет синхронизации: Процессы независимы, не нужны mutex/блокировки.

4.4 Какое количество процессов/потоков эффективнее всего

Оптимальное количество процессов/потоков равно числу физических ядер CPU, поскольку каждый поток/процесс эффективно использует отдельное ядро без конкуренции за вычислительные ресурсы. Большее число приводит к переключению контекста и снижению производительности из-за накладных расходов, а меньшее не использует все ядра.

5. Вывод

В ходе выполнения лабораторной работы были реализованы две программы умножения матриц размером N x N с использованием многопоточности и многопроцессорности. Обнаружено, что оптимальное количество потоков/процессов равно количеству физических ядер. При малом числе потоков наблюдается близкое к линейному ускорение благодаря эффективному использованию ядер и хорошей локальности данных в кэше, тогда как при числе превышающем количество физ. ядер производительность падает из за переключения контекста, конфликтов кэша и накладных расходов на управление. Процессы показали на 7-20% лучшую производительность по сравнению с потоками.

6. Приложение

matrix-proc.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <chrono>

using namespace std;

int main(int argc, char* argv[]) {
    int matrixSize;
    int numProcs;

    if (argc == 3) {
        matrixSize = atoi(argv[1]);
        numProcs = atoi(argv[2]);
    } else {
        cout << "Введите размер матрицы: ";
        cin >> matrixSize;
```

```
        cout << "Введите количество процессов: ";
        cin >> numProcs;
    }


    int size = matrixSize * matrixSize * sizeof(int);

    int* A = (int*)mmap(NULL, size, PROT_READ | PROT_WRITE,
                        MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    int* B = (int*)mmap(NULL, size, PROT_READ | PROT_WRITE,
                        MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    int* C = (int*)mmap(NULL, size, PROT_READ | PROT_WRITE,
                        MAP_SHARED | MAP_ANONYMOUS, -1, 0);


    srand(time(NULL));
    for (int i = 0; i < matrixSize * matrixSize; i++) {
        A[i] = rand() % 10 + 1;
        B[i] = rand() % 10 + 1;
        C[i] = 0;
    }

    auto start = chrono::high_resolution_clock::now();

    int rowsPerProc = matrixSize / numProcs;
    int extraRows = matrixSize % numProcs;
    int curRow = 0;

    for (int p = 0; p < numProcs; p++) {
        int startRow = curRow;
        int endRow = curRow + rowsPerProc;

        if (p < extraRows) {
```

```
            endRow++;
        }

        curRow = endRow;
        pid_t pid = fork();

        if (pid == 0) {
            for (int i = startRow; i < endRow; i++) {
                for (int j = 0; j < matrixSize; j++) {
                    C[i * matrixSize + j] = 0;
                    for (int k = 0; k < matrixSize; k++) {
                        C[i * matrixSize + j] += A[i *
matrixSize + k] * B[k * matrixSize + j];
                    }
                }
            }
            exit(0);
        }
    }

    for (int i = 0; i < numProcs; i++) {
        wait(NULL);
    }

    auto end = chrono::high_resolution_clock::now();
    auto duration =
chrono::duration_cast<chrono::duration<double, milli>>(end -
start);

    cout << "Размер матрицы: " << matrixSize << "x" <<
matrixSize << endl;
    cout << "Количество процессов: " << numProcs << endl;
```

```
    cout << "Время выполнения: " << duration.count() /
1000.0 << " секунд" << endl;

    if (matrixSize <= 5) {
        cout << "\nРезультат:" << endl;
        for (int i = 0; i < matrixSize; i++) {
            for (int j = 0; j < matrixSize; j++) {
                cout << C[i * matrixSize + j] << " ";
            }
            cout << endl;
        }
    }

    munmap(A, size);
    munmap(B, size);
    munmap(C, size);
}
```

matrix-threads.cpp

```cpp
#include <iostream>
#include <pthread.h>
#include <stdlib.h>
#include <chrono>

using namespace std;

int matrixSize;
int numThreads;
int** A;
int** B;
int** C;

struct Data {
    int startRow;
    int endRow;
};

void* multiply(void* arg) {
    Data* data = (Data*)arg;

    for (int i = data->startRow; i < data->endRow; i++) {
        for (int j = 0; j < matrixSize; j++) {
            C[i][j] = 0;
            for (int k = 0; k < matrixSize; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return NULL;
}
```

```
int main(int argc, char* argv[]) {
    if (argc == 3) {
        matrixSize = atoi(argv[1]);
        numThreads = atoi(argv[2]);
    } else {
        cout << "Введите размер матрицы: ";
        cin >> matrixSize;
        cout << "Введите количество потоков: ";
        cin >> numThreads;
    }

    A = new int*[matrixSize];
    B = new int*[matrixSize];
    C = new int*[matrixSize];
    for (int i = 0; i < matrixSize; i++) {
        A[i] = new int[matrixSize];
        B[i] = new int[matrixSize];
        C[i] = new int[matrixSize];
    }

    srand(time(NULL));
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixSize; j++) {
            A[i][j] = rand() % 10 + 1;
            B[i][j] = rand() % 10 + 1;
        }
    }

    auto start = chrono::high_resolution_clock::now();


    pthread_t* threads = new pthread_t[numThreads];
```

```
    Data* threadData = new Data[numThreads];

    int rowsPerThread = matrixSize / numThreads;
    int extraRows = matrixSize % numThreads;
    int curRow = 0;

    for (int i = 0; i < numThreads; i++) {
        threadData[i].startRow = curRow;
        threadData[i].endRow = curRow + rowsPerThread;

        if (i < extraRows) {
            threadData[i].endRow++;
        }

        curRow = threadData[i].endRow;

        pthread_create(&threads[i], NULL, multiply,
&threadData[i]);
    }

    for (int i = 0; i < numThreads; i++) {
        pthread_join(threads[i], NULL);
    }

    auto end = chrono::high_resolution_clock::now();
    auto duration =
chrono::duration_cast<chrono::duration<double, milli>>(end -
start);

    cout << "Размер матрицы: " << matrixSize << "x" <<
matrixSize << endl;
    cout << "Количество потоков: " << numThreads << endl;
```

```
    cout << "Время выполнения: " << duration.count() /
1000.0 << " секунд" << endl;

    if (matrixSize <= 5) {
        cout << "\nРезультат:" << endl;
        for (int i = 0; i < matrixSize; i++) {
            for (int j = 0; j < matrixSize; j++) {
                cout << C[i][j] << " ";
            }
            cout << endl;
        }
    }

    for (int i = 0; i < matrixSize; i++) {
        delete[] A[i];
        delete[] B[i];
        delete[] C[i];
    }
    delete[] A;
    delete[] B;
    delete[] C;
    delete[] threads;
    delete[] threadData;
}
```