Министерство цифрового развития, связи
и массовых коммуникаций Российской Федерации

Сибирский государственный университет
телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

ЛАБОРАТОРНАЯ РАБОТА №8

По дисциплине: «Операционные системы»

Выполнили:

Студенты 3 курса группы ИП-111
Корнилов А.А.,
Попов М.И.,
Толкач А.А.

Проверил:

Профессор кафедры ПМиК
Малков Е.А.

Новосибирск, 2023

Задание: программно реализуйте вычисление суммы последовательности чисел на основе последовательного кода, интерфейсов Pthreads и C++11 <thread>. Сравните время вычислений.

Цель: знакомство с программными интерфейсами управления потоками в Linux

Выполнение работы:

В качестве программы для вычисления суммы последовательности чисел была написана программа для вычисления числа Pi при помощи ряда Лейбца с точностью до 64 знаков после запятой (long double) на языках C (лист. 1) и C++ (лист. 2).

```c
#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <time.h>


int main() {
    clock_t start = clock();

    long double pi = 0;
    int j = 1;
    const int prec = 64;
    for (int i = 1; i < 500000000; i++) {
        pi += (double)4 / j;
        j += 2;
        pi -= (double)4 / j;
        j += 2;
    }

    printf("w = %2d p = %2d pi = %*.*Lf\n", 62, 62, 62, 62, pi);
    clock_t end = clock();
    printf("%s%f%s\n","Потрачено на вычисление и вывод: ",(double)(end - start) /
CLOCKS_PER_SEC, " сек.");
    return 0;
}
```

Листинг 1 – программа lab08_5.c

```cpp
#include <iostream>
#include <iomanip>
#include <time.h>
using namespace std;

int main(){
    clock_t start = clock();

    long double pi = 0;
    int j = 1;
    const int prec = 64;
    for (int i = 1; i < 500000000; i++){
        pi += (double)4 / j;
        j += 2;
```

```
        pi -= (double)4 / j;
        j += 2;
    }

    cout << setprecision(prec) << pi << endl;
    clock_t end = clock();
    cout << "Потрачено на вычисление и вывод: " << setprecision(3) << (double)(end -
start) / CLOCKS_PER_SEC << " сек." << endl;
    return 0;
}
```

Листинг 2 – программа lab08_6.cpp

Компиляция программы и вывод для C:

```
miron@DESKTOP-UMC1Q46:/mnt/u/Documents/B BУ3/OS/8$ gcc lab08_5.c -o lab08_5
miron@DESKTOP-UMC1Q46:/mnt/u/Documents/B BУ3/OS/8$ ./lab08_5
w=62 p=62
pi = 3.14159265258979539052995588743755206451169215142726898193359375
Потрачено на вычисление и вывод: 6.618006 сек.
```

Компиляция программы и вывод для C++:

```
miron@DESKTOP-UMC1Q46:/mnt/u/Documents/B BУ3/OS/8$ g++ lab08_6.cpp -o
lab08_6
miron@DESKTOP-UMC1Q46:/mnt/u/Documents/B BУ3/OS/8$ ./lab08_6
3.14159265258979539052995588743755206451169215142726898193359375
Потрачено на вычисление и вывод: 6.63 сек.
```

Позже в отчете будет приведена таблица (табл.1) для подсчетов точности, времени и правильности относительно реального числа pi.

Теперь модифицируем алгоритм программы lab08_5.c для параллельного вычисления при помощи библиотеки pthread.h (лист. 3). Вычисления будут проводиться на 4 потоках заданных NUM_THREADS и количество итераций ITERATIONS. Сама функция вычисления calculate_pi устроена так что делит количество вычисляемый итераций относительно числу потоков и при помощи mutex-ов осуществляться разделение доступа к числу pi от локального вычисляемого потоком local_pi.

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

```
#define NUM_THREADS 4
#define ITERATIONS 500000000

long double pi = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* calculate_pi(void* thread_id) {
    long double local_pi = 0;
    int thread_num = *(int*)thread_id;
    int start = thread_num * (ITERATIONS / NUM_THREADS) + 1;
    int end = (thread_num + 1) * (ITERATIONS / NUM_THREADS);

    for (int I = start; I <= end; i++) local_pi += (I % 2 == 1) ? (long double)4 / (2 * I
- 1) : -(long double)4 / (2 * I - 1);

    pthread_mutex_lock(&mutex);
    pi += local_pi;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main() {
    clock_t start = clock();
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    for (int I = 0; I < NUM_THREADS; i++) {
        thread_ids[i] = I;
        pthread_create(&threads[i], NULL, calculate_pi, (void*)&thread_ids[i]);
    }

    for (int I = 0; I < NUM_THREADS; i++) pthread_join(threads[i], NULL);

    printf("w = %2d p = %2d pi = %*.*Lf\n", 62, 62, 62, 62, pi);

    clock_t end = clock();
    printf("%s%f%s\n", "Потрачено на вычисление и вывод: ", (double)(end - start) /
CLOCKS_PER_SEC, " сек.");
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Листинг 3 – программа lab08_5p.cpp

Компиляция программы и вывод:

```
miron@DESKTOP-UMC1Q46:/mnt/u/Documents/B BY3/OS/8$ gcc lab08_5p.c -lpthread
-o lab08_5p
miron@DESKTOP-UMC1Q46:/mnt/u/Documents/B BY3/OS/8$ ./lab08_5p
w = 62 p = 62 pi =
3.14159265158979430143312838730196290271123871207237243652343750
Потрачено на вычисление и вывод: 3.410105 сек.
```

Время вычисления сократилось вдвое.

Также модифицируем программу lab08_6.c написанную на языке C++, для использования параллельности будем использовать библиотеку thread (лист. 4). Вместо стандартной переменной long double будем использовать атомарную переменную типа double (методы для long double по неизвестной причине не доступны), барьерное разделение вычислений теперь будет проходить в main функции программы по такому-же принципу.

```
#include <iostream>
#include <iomanip>
#include <thread>
#include <vector>
#include <atomic>
#include <ctime>
using namespace std;

const int num_threads = 4; // Количество потоков для параллельных вычислений
const int iterations_per_thread = 500000000; // Количество итераций на каждом потоке

atomic<double> pi{ 0.0 }; // Используем атомарную переменную для суммирования

void calculatePi(int start, int end) {
    double localSum = 0;
    int j = 2 * start + 1;

    for (int I = start; I < end; i++){
        if (I % 2 == 0) {
            localSum += 4.0 / j;
        }
        else {
            localSum -= 4.0 / j;
        }
        j += 2;
    }
    pi.store(localSum, memory_order_relaxed);
}

int main() {
    clock_t start_time = clock();
    vector<thread> threads;
    for (int I = 0; I < num_threads; i++) {
        int start = I * iterations_per_thread;
        int end = start + iterations_per_thread;
        threads.push_back(thread(calculatePi, start, end));
    }

    for (auto& t : threads) t.join();

    cout << setprecision(64) << pi << endl;
    clock_t end_time = clock();
    cout << "Потрачено на вычисление и вывод: " << setprecision(3) << (double)(end_time -
start_time) / CLOCKS_PER_SEC << " сек." << endl;
    return 0;

}
```

Компиляция программы и вывод:

```
miron@DESKTOP-UMC1Q46:/mnt/u/Documents/B BY3/OS/8$ g++ lab08_6p.cpp -o
lab08_6p
miron@DESKTOP-UMC1Q46:/mnt/u/Documents/B BY3/OS/8$ ./lab08_6p
3.14159264558932393995860365976113826036453247070312 5
Потрачено на вычисление и вывод: 1.64 сек.
```

Время работы сократилось в 4-е раза, но программа не всегда выдает точный результат, этот был получен с 6-ой попытки.

В таблице (табл. 1) приведены все подсчеты программы с учетом использование параллельности и использованием разных компиляторов. Точность указывается от количества полученных в ответе чисел, в расхождении указывается после какого числа начинаться отличия по сравнением с оригинальным числом.

Компилятор	Файл	Время выполнения, сек.	Результат	Точность	Расхождение
Оригинал числа с Wiki			3.14159265358979323846264338327950288419716939937510582097494459	63	63
GCC	lab08_5.c	6.62	3.14159265258979539052995588743755206451169215142726898193359375	64	15
G++	lab08_6.c	6.63	3.14159265258979539052995588743755206451169215142726898193359375	63	15
MVS CL	lab08_6.c	3.24	3.14159265258805042719814082374796271324157714843 75	50	9
GCC	lab08_5p.c	3.41	3.14159265158979430143312838730196290271123871207237243652343750	63	9
G++	lab08_6p.c	1.64	3.14159264558932393995860365976113826036453247070312 5	52	8
MVS CL	lab08_6p.c	0.467	3.14159264558932393995860365976113826036453247070312 5	52	9

Таблица 1 – Результаты работ программы и их подсчеты

Вывод: С разделением программ на для параллельного вычисления точность вычислений сильно падает, быстрейшими компилятором для программ оказался MVS CL. Вычисления проводились с Windows 11 и оболочной WSL.