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»

Группа: М8О-214БВ-24

Студент: Чернявская Алиса Алексеевна

Преподаватель: Бахарев В.Д. (ФИИТ)

Оценка: _____

Дата: 03.10.25

Москва, 2025

Постановка задачи

Вариант 5.

Задание

Пользователь вводит команды вида: «число». Далее это число передается от родительского процесса в дочерний. Дочерний процесс производит проверку на простоту. Если число составное, то в это число записывается в файл. Если число отрицательное или простое, то тогда дочерний и родительский процессы завершаются.

Программу выполнить с помощью использования семафоров для синхронизации и shared memory.

Общий метод и алгоритм решения

Использованные системные вызовы:

CreateThread() — создание нового потока

• WaitForSingleObject() — ожидание завершения потока

• CloseHandle() — закрытие handles

• CreateMutex() — создание мьютекса

• WaitForSingleObject() — захват мьютекса (аналог wait в unix)

• ReleaseMutex() — освобождение мьютекса (уменьшение счетчика)

• GetSystemInfo() — получение количества логических процессоров

• GetCurrentThreadId() — используется для инициализации «случайного» числа

• WriteFile() — запись консоль (аналог вместо printf)

• GetStdHandle() — получение handle стандартного вывода

В программе реализованы 2 способа расчета вероятности того, что в колоде из 52 карт сверху лежат две одинаковые: последовательный и параллельный. В последовательном алгоритме все вычисления выполняются друг за другом, без синхронизации. Алгоритм работы последовательной версии: получение числа раундов (т.е. итераций, беру 1000000), генерация двух карт значениями от 0 до 12 и их сравнение. Дальше алгоритм считает число совпадений на этих итерациях и замеряет время выполнения всех итераций в микросекундах. После этого выводится вычисленная вероятность.

Как работает параллельный алгоритм: сначала создаем память под массив потоков и их аргументов (потоки представлены структурами). Затем насколько возможно равномерно распределяется количество раундов между потоками (число потоков прописано в main). Для синхронизации общего счётчика совпадений создаётся мьютекс. Дальше каждый поток выполняет такую функцию: генерирует карты, подсчитывает совпадения на том количестве раундов, которое ему досталось и добавляет число совпадений к общему счетчику через мьютекс. Главный поток ожидает завершения каждого дочернего потока. После завершения всех потоков измеряется время выполнения и выводится итоговая вероятность.

Параллельный расчет выполняется для разных количеств ядер: 1, 2, 3, 4 (как и количество процессоров в операционной системе), 32, 128, 256.

Код программы

```
#include <windows.h>

// структура для потока
typedef struct
{
    SIZE_T thread_id;
    SIZE_T numrounds;// количество раундов на поток
    LONGLONG success;// счетчик успешных вариантов для каждого потока
    DWORD rand;// число для рандомайзера
} ThreadArgs;

static LONGLONG total = 0;// счетчик успешных вариантов суммарно для всех потоков
static HANDLE mutex;

//функция для вывода в консоль строки
void strprint(const char* str)
{
    HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hStdout == INVALID_HANDLE_VALUE)
    {
        return;
    }
    DWORD written;
    WriteFile(hStdout, str, lstrlenA(str), &written, NULL);
}

// функция вывода числа в консоль
void numprint(SIZE_T num)
{
    char buffer[32];
    int pos = 31;
    buffer[31] = '\0';
    if (num == 0)
    {
        buffer[--pos] = '0';
    }
    else
    {
        while (num > 0)
        {
            buffer[--pos] = '0' + (num % 10);
            num /= 10;
        }
    }
    strprint(&buffer[pos]);
}

// преобразование строки в число
SIZE_T strtoint(const char* str)
{
    SIZE_T result = 0;
    while (*str >= '0' && *str <= '9')
    {
        result = result * 10 + (*str - '0');
        str++;
    }
    return result;
}

// вывод процентов
void printchance(double prob)
{
    int percent = (int)(prob * 100);//целое
```

```
    int decimals = (int)(prob * 10000) % 100;//дробь
    strprint("chance: ");
    numprint(percent);
    strprint(".");
    if (decimals < 10)
        strprint("0");
    numprint(decimals);
    strprint("%");
    strprint("\n");
}

//генератор рандомных чисел для генерации карточек
DWORD random(DWORD* rand)
{
    return (*rand = *rand * 123 + 456);
}

// проверка 2х верхних карт
DWORD cards(DWORD* rand)
{
    DWORD card1 = random(rand) % 13;
    DWORD card2 = random(rand) % 13;
    if (card1 == card2)
        return 1;
    else
        return 0;
}

// последовательный алгоритм!!!! - здесь нет потоков и все выполняется сразу
void one(SIZE_T rounds)
{
    DWORD rand = GetTickCount();
    LONGLONG well = 0;//счетчик успешных успехов
    DWORD start = GetTickCount();
    // Основной цикл - обрабатываем все раунды в одном потоке
    for (SIZE_T i = 0; i < rounds; i++)
    {
        well += cards(&rand);
    }
    DWORD time = GetTickCount() - start;
    strprint("long algorithm: ");
    numprint((SIZE_T)time);
    strprint(" ms\n");
    double chance = (double)well / (double)rounds;
    printchance(chance);
}

// функция для каждого потока для последовательной реализации
DWORD WINAPI Thread(LPVOID args)
{
    ThreadArgs* structarg = (ThreadArgs*)args;
    structarg->success = 0;
    structarg->rand = GetCurrentThreadId();//делаем число для рандома рандомным
    // обновление счетчика в случае успеха
    for (SIZE_T i = 0; i < structarg->numrounds; i++)
    {
        structarg->success += cards(&structarg->rand);
    }
    WaitForSingleObject(mutex, INFINITE);//МЬЮТЕКС чтобы общий счетчик не прибавлялся у всех
сразу
    total += structarg->success;
    ReleaseMutex(mutex); // освобождение мьютекса для следующего потока
    return 0;
}

LONGLONG micros()
{
    static LARGE_INTEGER freq = { 0 };
    LARGE_INTEGER counter;
```

```
    if (freq.QuadPart == 0)
        QueryPerformanceFrequency(&freq);

    QueryPerformanceCounter(&counter);
    return (counter.QuadPart * 1000000LL) / freq.QuadPart; // в микросекундах
}

//параллельный алгоритм!!! в нем много потоков
void parallel(SIZE_T rounds, SIZE_T numthreads)
{
    total = 0;

    HANDLE* threads = HeapAlloc(GetProcessHeap(), 0, numthreads * sizeof(HANDLE));
    ThreadArgs* structarg = HeapAlloc(GetProcessHeap(), 0, numthreads * sizeof(ThreadArgs));

    if (!threads || !structarg) {
        strprint("memory allocation error\n");
        return;
    }

    SIZE_T thread_round = rounds / numthreads;
    SIZE_T ost = rounds % numthreads;

    LONGLONG start = micros();

    for (SIZE_T i = 0; i < numthreads; i++) {
        structarg[i].thread_id = i;
        structarg[i].numrounds = thread_round + (i < ost ? 1 : 0);
        threads[i] = CreateThread(NULL, 0, Thread, &structarg[i], 0, NULL);

        if (threads[i] == NULL) {
            strprint("error creating thread\n");
            numprint(i);
            strprint("\n");
            numthreads = i; // ждём только то, что создали
            break;
        }
    }

    // 👇 БЕЗ ОГРАНИЧЕНИЯ 64 потоков
    for (SIZE_T i = 0; i < numthreads; i++) {
        WaitForSingleObject(threads[i], INFINITE);
        CloseHandle(threads[i]);
    }

    LONGLONG end = micros();
    LONGLONG duration = end - start;

    strprint("thread number ");
    numprint(numthreads);
    strprint(" - ");
    numprint(duration);
    strprint(" microseconds\n");

    double chance = (double)total / (double)rounds;
    printchance(chance);

    HeapFree(GetProcessHeap(), 0, threads);
    HeapFree(GetProcessHeap(), 0, structarg);
}



int main(int argc, char* argv[])
{
    SYSTEM_INFO sysinfo;
    GetSystemInfo(&sysinfo);
    size_t cernel = sysinfo.dwNumberOfProcessors;
```

```
    SIZE_T rounds = 1000000;
    if (argc > 1)
    {
        rounds = strtoint(argv[1]);
    }
    //запуск последовательного алгоритма
    one(rounds);

    mutex = CreateMutex(NULL, FALSE, NULL);
    if (mutex == NULL)
    {
        strprint("cannot create mutex\n");
        return 1;
    }

    parallel(rounds, 1);
    parallel(rounds, 2);
    parallel(rounds, 3);
    strprint("the next thread is the number of cernels - ");
    numprint(cernel);
    strprint("\n");
    parallel(rounds, cernel);
    parallel(rounds, 32);
    parallel(rounds, 128);
    parallel(rounds, 256);

    CloseHandle(mutex);

    return 0;

}
```

Протокол работы программы

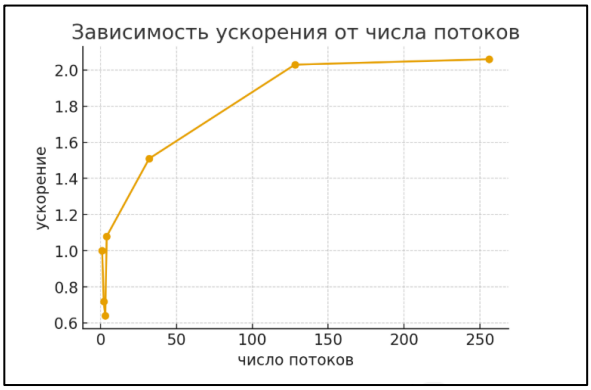
Сборка:

```
C:\Users\Алиска\Documents\mai\osi\1122>program.exe 100000000
long algorithm: 1265 ms
chance: 7.69%
thread number 1 - 1303014 microseconds
chance: 7.69%
thread number 2 - 1813866 microseconds
chance: 7.69%
thread number 3 - 2028164 microseconds
chance: 7.69%
the next thread is the number of cernels - 4
thread number 4 - 1203473 microseconds
chance: 7.69%
thread number 32 - 862096 microseconds
chance: 7.69%
thread number 128 - 643008 microseconds
chance: 7.68%
thread number 256 - 633521 microseconds
chance: 7.68%
```

Объяснение полученных результатов:

Один процесс выполняет работу быстрее, чем несколько процессов сразу, потому что ему нужно только сгенерировать 2 карты и выполнить одно сравнение для них. Эта работа занимает микросекунды, а обращение к ядру, создание потоков, ожидание, захват и высвобождение мьютексов занимают миллисекунды. Если бы нужно было генерировать 1млн карт, тогда скорее всего ускорение было бы заметно, но поскольку задание является простым в плане реализации, многопоточность бессмысленна и ускорения не наблюдается. Поэтому у последовательного алгоритма (который работает как 1 поток), у однопоточного и у количества потоков, равного числу процессоров системы время примерно одинаковое. При этом самый эффективный из них – с количеством потоков, равных количеству процессоров, так как получается, что каждый поток получает по ядру и не тратится время на переключение контекстов и на ожидание всех потоков. Для большого же числа потоков очень много времени тратится на гонку потоков за ядра, на обращения к ним, на переключение контекстов и на ожидание всех процессов, поэтому это не самая выгодная и эффективная стратегия для такой задачи, однако ускорение наблюдается (причем чем больше ядер, тем выше ускорение), поскольку каждый поток работает микросекунды и часть потоков успевает закончить еще до того, как другие были созданы. Эффективность же показывает, сколько пользы дает 1 поток. Так как при большом числе потоков ускорение растет за счет очень большого количества обращения к ядру, производительность падает.

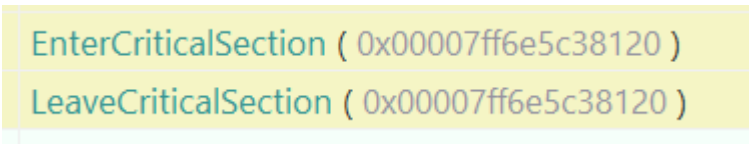
число потоков	время исполнения	ускорение	эффективность
1	1303014	1	1
2	1813866	0,72	0,359
3	2028164	0,64	0,214
4	1203473	1,08	0,271
32	862096	1,51	0,047
128	643008	2,03	0,016
256	633521	2,06	0,008



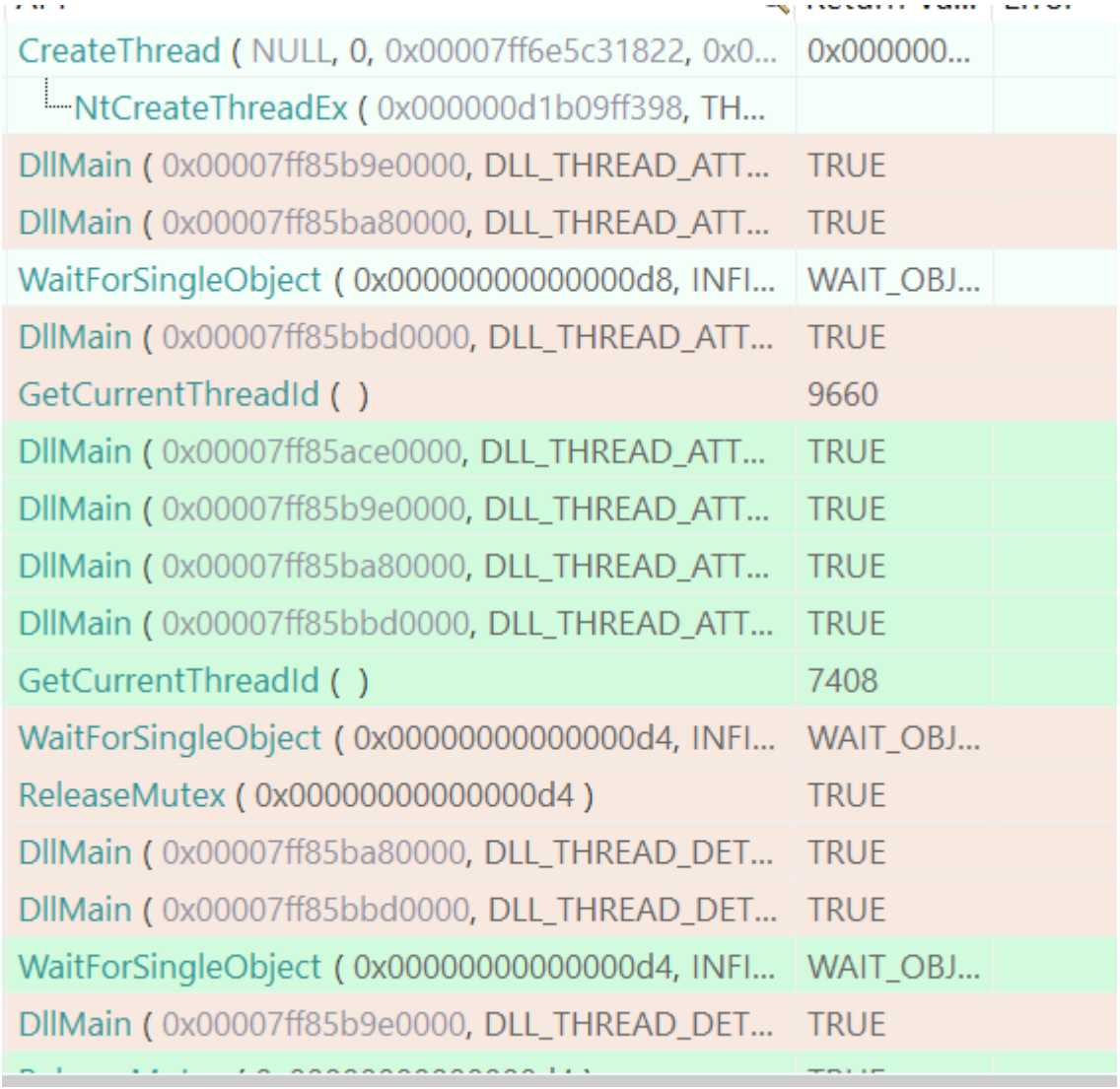
Вывод системных вызовов через API monitor:

Создание потоков и захват-освобождение мьютекса:

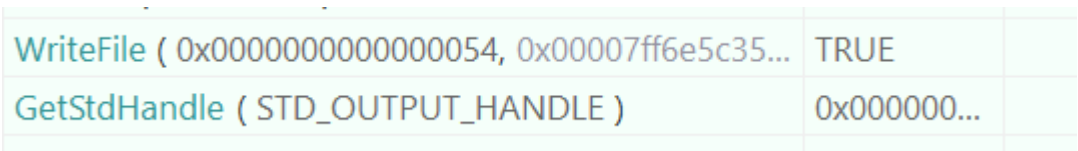
Вход – выход из критической позиции доступа всех потоков к одним данным:



Еще создание потоков и мьютексы на всякий случай:



Вывод в консоль:



Вывод

В ходе лабораторной работы была реализована многопоточная программа на языке C с использованием системных механизмов Windows для параллельной обработки данных. Были применены базовые средства работы с потоками, включая создание потоков через CreateThread, синхронизацию доступа к общим данным с помощью мьютекса (CreateMutex, WaitForSingleObject, ReleaseMutex), а также высокоточное измерение времени посредством QueryPerformanceCounter. Программа продемонстрировала сравнение последовательного и

параллельного выполнения, оценку ускорения и эффективности при различном числе потоков, а также особенности работы планировщика Windows при высокой степени параллелизма. Все выделенные ресурсы были корректно освобождены, что исключает утечки памяти и обеспечивает корректное завершение программы.

Максимальная эффективность была достигнута при выполнении программы одним потоком. При этом максимальное ускорение наблюдается при количестве процессов равном 256.

Самое короткое время выполнения – на последовательной реализации алгоритма и при количестве процессов равных количеству процессоров. В целом, задание слишком простое для того, чтобы использовать многопоточное выполнение.