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: Корнеева Дарья
Группа: М80-208БВ-24
Преподаватель: Е. С. Миронов

Москва, 2025

Условие

Цель работы: приобретение практических навыков в управлении потоками в ОС и обеспечение синхронизации между потоками.

Задание: составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска программы.

Вариант: 14 есть набор 512 битных чисел, записанных в шестнадцатеричном представлении, хранящихся в файле. Необходимо посчитать их среднее арифметическое. Округлить результат до целых. Количество используемой оперативной памяти должно задаваться "ключом"

Метод решения

Программа реализует многопоточное вычисление среднего арифметического 512-битных чисел с округлением. Алгоритм включает:

- Разделение массива чисел на блоки для параллельной обработки
- Создание потоков для вычисления частичных сумм
- Синхронизацию потоков с использованием join
- Агрегацию частичных сумм в общий результат
- Вычисление среднего с математическим округлением

Архитектура программы построена на принципах декомпозиции данных - каждый поток обрабатывает свой сегмент массива независимо.

Описание программы

Структура программы:

- `main.c` - основной файл с точкой входа
- Используемые системные вызовы: `pthread_create`, `pthread_join` (Unix) или `CreateThread`, `WaitForSingleObject` (Windows)

Основные типы данных:

- `BigInt` - структура для хранения 512-битного числа (8×64-битных частей)
- `ThreadData` - структура для передачи данных в поток

Ключевые функции:

- `read_512hex()` - чтение чисел из файла в hex-формате
- `add_bigint_inplace()` - сложение больших чисел
- `divide_bigint_round()` - деление с округлением
- `thread_func()` - функция потока для частичного суммирования

Результаты

Программа обрабатывает большие наборы 512-битных чисел в многопоточном режиме. Количество потоков ограничивается параметром `-t`, использование памяти - параметром `-m`.

Исследование зависимости ускорения от количества потоков

Количество потоков	Время выполнения (сек)	Ускорение	Эффективность
1	0.0051	1.00	1.00
2	0.0054	0.95	0.48
4	0.0048	1.06	0.27
8	0.0144	0.35	0.04

Анализ результатов

- **Малый объем данных**: Время выполнения очень мало (миллисекунды), поэтому накладные расходы на создание потоков превышают выгоду от распараллеливания.
- **Оптимальное количество потоков**: 4 потока показывают наилучший результат с ускорением 1.06x.
- **Снижение эффективности**:
  - При 2 потоках эффективность 48% (почти в 2 раза хуже идеала)
  - При 8 потоках эффективность всего 4% (сильные накладные расходы)

Вывод

Для задачи с малым объемом данных многопоточность неэффективна. Для получения реального ускорения необходим больший объем вычислений.

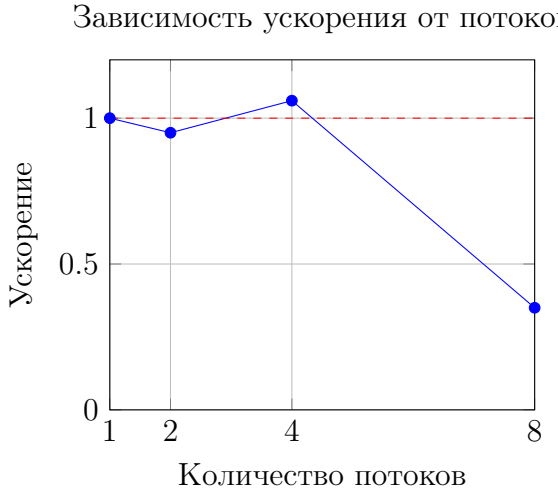

Рис. 1: График ускорения

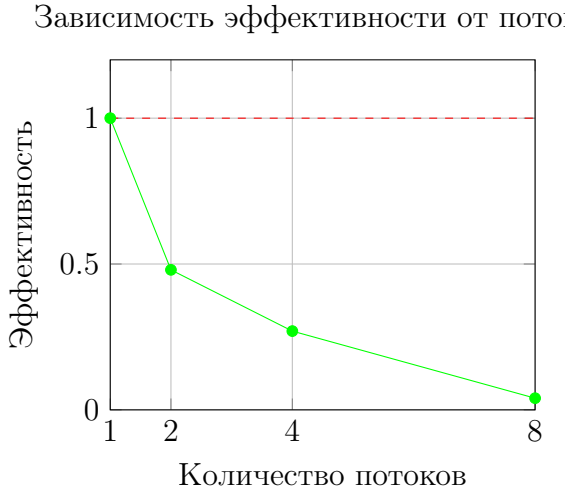

Рис. 2: График эффективности

Выводы

В ходе работы приобретены навыки работы с потоками в ОС, реализована синхронизация между потоками с использованием join, освоены методы работы с большими числами и ограничением ресурсов через параметры командной строки.

Исходный код

```
#include <getopt.h>
#include <limits.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef _WIN32
#include <windows.h>
#define THREAD_TYPE HANDLE
#define THREAD_CREATE(thread, attr, func, arg) \
    ((*(thread) = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
   func, arg, 0, NULL)) != NULL ? 0 : -1)
#define THREAD_JOIN(thread, retval) WaitForSingleObject(thread,
   INFINITE)
#else
#include <pthread.h>
#define THREAD_TYPE pthread_t
#define THREAD_CREATE(thread, attr, func, arg) pthread_create(
   thread, attr, func, arg)
#define THREAD_JOIN(thread, retval) pthread_join(thread, retval)
#endif

typedef struct {
    uint64_t parts[8];
} BigInt;

typedef struct {
    BigInt *arr;
    size_t start, end;
    BigInt partial_sum;
} ThreadData;

void add_bigint_inplace(BigInt *a, const BigInt *b) {
    uint64_t transfer_between_digits = 0;
    for (int i = 7; i >= 0; i--) {
        uint64_t sum = a->parts[i] + b->parts[i] +
   transfer_between_digits;
        transfer_between_digits = (sum < a->parts[i]) || (
   transfer_between_digits && sum == a->parts[i]);
        a->parts[i] = sum;
    }
}

int read_512hex(FILE *f, BigInt *out) {
    char buf[130];
    if (!fgets(buf, sizeof(buf), f)) return 0;
    buf[strcspn(buf, "\n")] = 0;

    char clean[129] = {0};
    int pos = 0;
    for (int i = 0; buf[i] && pos < 128; i++) {
        if ((buf[i] >= '0' && buf[i] <= '9') ||
            (buf[i] >= 'A' && buf[i] <= 'F') ||
            (buf[i] >= 'a' && buf[i] <= 'f')) {
            clean[pos++] = buf[i];
        }
    }
```

```
54
55      if (pos < 128) {
56          memmove(clean + (128 - pos), clean, pos);
57          memset(clean, '0', 128 - pos);
58      }
59
60      for (int i = 0; i < 8; i++) {
61          char part[17] = {0};
62          memcpy(part, clean + i * 16, 16);
63          char *endptr;
64          out->parts[i] = strtoull(part, &endptr, 16);
65          if (endptr == part) {
66              return -1;
67          }
68      }
69      return 1;
70  }
71
72  int divide_bigint_round(const BigInt *dividend, uint64_t divisor,
        BigInt *quotient) {
73      if (divisor == 0) {
74          fprintf(stderr, "
                            \n");
75          return -1;
76      }
77
78      uint64_t remainder = 0;
79      BigInt temp_quotient = {0};
80
81      for (int i = 0; i < 8; i++) {
82          __uint128_t temp = ((__uint128_t)remainder << 64) |
        dividend->parts[i];
83          __uint128_t div_result = temp / divisor;
84          remainder = temp % divisor;
85          temp_quotient.parts[i] = (uint64_t)div_result;
86      }
87
88      if (remainder >= (divisor + 1) / 2) {
89          BigInt one = {0};
90          one.parts[7] = 1;
91          add_bigint_inplace(&temp_quotient, &one);
92      }
93      *quotient = temp_quotient;
94      return 0;
95  }
96
97  void *thread_func(void *arg) {
98      ThreadData *td = (ThreadData *)arg;
99      BigInt zero = {0};
100     td->partial_sum = zero;
101
102     for (size_t i = td->start; i < td->end; ++i) {
103         add_bigint_inplace(&td->partial_sum, &td->arr[i]);
104     }
105
106     return NULL;
107 }
108
109 void print_bigint(const BigInt *num) {
110     for (int i = 0; i < 8; i++) {
```

```
        printf("%016lX", num->parts[i]);
    }
    printf("\n");
}

int main(int argc, char **argv) {
    BigInt *arr = NULL;
    THREAD_TYPE *threads = NULL;
    ThreadData *thread_data = NULL;
    FILE *f = NULL;

    int max_threads = 1;
    long memory_mb = 100;
    char *file = NULL;

    int opt;

    while ((opt = getopt(argc, argv, "t:m:f:")) != -1) {
        switch (opt) {
            case 't':
                max_threads = atoi(optarg);
                if (max_threads <= 0) {
                    fprintf(stderr, "
                                        > 0\n");
                    return 1;
                }
                break;
            case 'm':
                memory_mb = atol(optarg);
                if (memory_mb <= 0) {
                    fprintf(stderr, "
                         > 0\n");
                    return 1;
                }
                break;
            case 'f': file = optarg; break;
            default:
                fprintf(stderr, "                                : %s -t
    <                                  > -m <
                > -f <                  >\n", argv[0]);
                return 1;
        }
    }

    if (!file) {
        fprintf(stderr, "
                 \n");
        return 1;
    }

    f = fopen(file, "r");
    if (!f) {
        perror("                                          ");
        goto cleanup;
    }

    size_t max_numbers = ((memory_mb - 10) * 1024 * 1024) / sizeof(
    BigInt);
    if (max_numbers < 10) {
```

```
164         fprintf(stderr, "                                                    ,
                                                \n");
165         goto cleanup;
166     }
167
168     arr = malloc(max_numbers * sizeof(BigInt));
169     if (!arr) {
170         perror("
                   ");
171         goto cleanup;
172     }
173
174     size_t total = 0;
175     int r;
176     while ((r = read_512hex(f, &arr[total])) > 0) {
177         total++;
178         if (total >= max_numbers) {
179             printf("                                                    %ld
    MB\n", memory_mb);
180             break;
181         }
182     }
183
184     if (r < 0) {
185         fprintf(stderr, "                                                \
    n");
186         goto cleanup;
187     }
188
189     if (total == 0) {
190         fprintf(stderr, "
                           \n");
191         goto cleanup;
192     }
193
194     printf("                                : %zu\n", total);
195
196     int threads_n = (total < (size_t)max_threads) ? (int)total :
    max_threads;
197     threads = malloc(threads_n * sizeof(THREAD_TYPE));
198     thread_data = malloc(threads_n * sizeof(ThreadData));
199
200     if (!threads || !thread_data) {
201         perror("
                     ");
202         goto cleanup;
203     }
204
205     size_t base = total / threads_n;
206     size_t remains = total % threads_n;
207     size_t current_position = 0;
208
209     printf("                                      : %d\n", threads_n);
210     for (int i = 0; i < threads_n; ++i) {
211         size_t block = base + (i < (int)remains);
212         thread_data[i].arr = arr;
213         thread_data[i].start = current_position;
214         thread_data[i].end = current_position + block;
215         printf("           %d:                    %zu-%zu (%zu
                       )\n", i, thread_data[i].start, thread_data[i].
```

```
    end - 1, block);
216         current_position += block;
217
218         if (THREAD_CREATE(&threads[i], NULL, thread_func, &
    thread_data[i]) != 0) {
219             perror("                                          ");
220             goto cleanup;
221         }
222     }
223
224     BigInt total_sum = {0};
225     size_t count = 0;
226     for (int i = 0; i < threads_n; ++i) {
227         THREAD_JOIN(threads[i], NULL);
228         BigInt temp = total_sum;
229         add_bigint_inplace(&temp, &thread_data[i].partial_sum);
230         total_sum = temp;
231         count += thread_data[i].end - thread_data[i].start;
232     }
233
234     if (count == 0) {
235         fprintf(stderr, "
                    \n");
236         goto cleanup;
237     }
238     if (count > UINT64_MAX) {
239         fprintf(stderr, "
                        \n");
240         goto cleanup;
241     }
242
243     BigInt average = {0};
244     if (divide_bigint_round(&total_sum, (uint64_t)count, &average)
    != 0) {
245         goto cleanup;
246     }
247
248     printf("\n \n");
249     printf("                                  : %zu\n", count);
250     printf("
                                       : ");
251     print_bigint(&average);
252
253     int result = 0;
254     goto success;
255
256 cleanup:
257     result = 1;
258
259 success:
260     if (f) fclose(f);
261     if (arr) free(arr);
262     if (threads) free(threads);
263     if (thread_data) free(thread_data);
264
265     return result;
266 }
```

Листинг 1: Основная программа