МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра Програмної інженерії

Звіт
з лабораторної роботи №7
з дисципліни: «Операційні системи»
з теми: «Керування процесами та потоками»

Виконали:
ст. гр. ПЗПІ-23-2
Ситник Є. С.
Малишкін. А. С.

Перевірила:
доц. каф. ПІ,
Мельнікова Р. В.,

Харків – 2025

7 КЕРУВАННЯ ПРОЦЕСАМИ ТА ПОТОКАМИ

7.1 Мета роботи

Метою даної лабораторної роботи є вивчення створення процесів та потоків при виконанні програм.

7.2 Хід роботи

Дану лабораторну роботу нами було виконано для операційної системи Linux із використанням POSIX API.

7.2.1 Завдання 1 – програмний комплекс для створення та аналізу текстових файлів.

Метою цього завдання є дослідження створення дочірніх процесів із своєї програми.

Завдання полягає у розробці трьох програм:

а) програма 1 запускає текстовий редактор у заданій папці;

б) програма 2 аналізує файли у заданій директорії, створені після вказаного часу,

визначаючи для них розмір, кількість рядків та довжину кожного рядка, з підтримкою ASCII та UNICODE кодувань;

а) програма 3 запускає спочатку програму 1, а потім програму 2, передаючи час запуску

першої програми як параметр для другої.

7.2.1.1 Розробка програми 1

Програма 1 повинна запустити системний текстовий редактор за замовчуванням у вказаній директорії.

Для цього необхідно:

а) визначити редактор за замовчуванням;

б) створити вказану директорію, якщо вона не існує;

в) запустити редактор у вказаній директорії.

Назва текстового редактору за замовчуванням на POSIX сумісних операційних системах зазвичай зберігається в змінній оточення «EDITOR», отримати її значення можна за допомогою функції «getenv» стандартної бібліотеки. Ця функція повертає вказівник на рядок, що відповідає значенню змінної, або «NULL», якщо ця змінна відсутня. Скористаємося цим, щоб встановити редактор за замовчуванням.

Щоб визначити чи директорія для файлів існує скористаємося системним викликом «stat» – якщо він завершиться із помилкою, значить директорії не існує, та її треба створити. Для створення директорії в разі її відсутності використаємо системний виклик «mkdir».

В POSIX сумісних операційних системах для запуску зовнішніх програм використовується системний виклик «exec». Цей системний виклик замінює образ поточного процесу на новий, а не створює новий процес, тому щоб відкрити редактор у вказаній директорії необхідно зміни робочу директорію поточного процесу. Для цього скористаємося системним викликом «chdir».

```c
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include "shared.h"

int main() {
  char *editor = getenv("EDITOR");
  if (editor == NULL)
    editor = "vi";

  struct stat st = {0};
  if (stat(CWD, &st) == -1) {
    if (mkdir(CWD, S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH) == -1) {
      err("Can't create directory for files", __LINE__ - 1);
      return 1;
    }
  }

  if (chdir(CWD) == -1) {
    err("Can't change working directory", __LINE__ - 1);
    return 2;
  }

  if (execlp(editor, editor, (char *)NULL)) {
    err("Can't launch editor", __LINE__ - 1);
    return 3;
  }

  return 0;
}
```

7.2.1.2 Розробка програми 2

Друга програма в якості аргументу отримує час, файли відредаговані після якого необхідно обробити. Перевіримо кількість аргументів, що були передані програмі, та перетворимо перший з них в потрібний формат, для цього скористаємося функцією «strptime» стандартної бібліотеки.

```
if (argc > 2) {
  fprintf(stderr, "Error: too many arguments.\n");
  return 1;
}

if (argc < 2) {
  fprintf(stderr, "Error: too few arguments.\n");
  return 1;
}

struct tm tm = {0};
if (strptime(argv[1], "%s", &tm) == NULL) {
  fprintf(
    stderr,
    "Error: please provide time in UNIX timestamp format.\n"
  );
  return 1;
}
time_t norm_time = mktime(&tm);
```

Для зручного перебору всіх файлів в директорії скористаємося засобами «FTS»[1] , що надає стандарт POSIX.

Створимо об’єкт FTS із коренем в директорії для файлів.

```
char *paths[] = {CWD, NULL};
FTS *fts = fts_open(paths, FTS_NOCHDIR, NULL);
if (fts == NULL) {
  err("Can't initialise FTS", __LINE__ - 1);
  return 2;
}
```

Виділимо буфер, в який будемо зберігати вміст файлів.

```
char *buf = (char *)malloc(1024 * sizeof(char));
```

[1]FTS (File Tree Traversal) в POSIX — це набір функцій, що дозволяють послідовно обходити файлову ієрархію (директорії та файли). FTS надає структурований спосіб для навігації по файловій системі, отримуючи інформацію про кожен знайдений елемент.

Та обробимо всі файли, що були змінені після вказаного часу, не заглиблюючись на інші рівні ієрархії файлової системи.

```
for (FTSENT *ent = fts_read(fts); ent != NULL; ent = fts_read(fts)) {
  switch (ent->fts_info) {
  case FTS_F:
    if (ent->fts_statp->st_mtime > norm_time) {
      ...
    }
    break;
  default:
    break;
  }
}

free(buf);
```

Відкриємо кожен файл та визначимо його кодування (UTF-8, UTF-16LE, UTF-16BE чи CP1251).

```
FILE *file = fopen(ent->fts_path, "r");
if (file == NULL) {
  err("Can't open the file", __LINE__ - 1);
  break;
}

size_t bytes_read = fread(buf, 1, 1023, file);
if (ferror(file) != 0) {
  err("Can't read the file", __LINE__ - 2);
  fclose(file);
  break;
}
buf[bytes_read] = '\0';

Encoding enc = get_encoding(buf, bytes_read);
```

В залежності від кодування порахуємо символи нових рядків в файлі. Для UTF-8 та CP1251 новий рядок позначається як один байт «\n», а для UTF-16LE та UTF-16BE як 2 байти «\n\0» та «\0\n» відповідно.

```
printf("Lines length:");
size_t lines = 0;
size_t line_len = 0;
for (size_t i = (enc == 2 || enc == 1) ? 2 : 0; i < bytes_read; i++)
{
  line_len++;

  int newline_detected = 0;
  switch (enc) {
  case ENC_UTF16BE:
    if (i + 1 < bytes_read && buf[i] == '\0' && buf[i + 1] == '\n') {
```

```
            newline_detected = 1;
            line_len++;
            i++;
          }
          break;
        case ENC_UTF16LE:
          if (i + 1 < bytes_read && buf[i] == '\n' && buf[i + 1] == '\0') {
            newline_detected = 1;
            line_len++;
            i++;
          }
          break;
        case ENC_CP1251:
        case ENC_UTF8:
        default:
          if (buf[i] == '\n')
            newline_detected = 1;
          break;
        }

        if (newline_detected) {
          printf(" %zu", line_len);
          lines++;
          line_len = 0;
        }
      }
      printf("\nTotal lines: %zu\n", lines);
```

Надрукуємо вміст файлу в консоль. Для зручнішого друку якщо текст має будь-яке кодування крім UTF-8 перетворимо його в UTF-8 скориставшись засобами «iconv»[2] , що надає стандарт POSIX.

```
if (enc != ENC_UTF8)
  buf = to_utf8(buf, bytes_read, enc);

printf(NORMAL);
printf("%s\n", buf);
```

7.2.1.3 Розробка програми 3

Третя програма повинна по черзі запустити 2 інші, при цьому зберігши час запуску першої, та передавши його другій.

```
time_t t = time(NULL);
char s[256];
strftime(s, 256, "%s", localtime(&t));
```

[2]iconv в POSIX – це бібліотека, яка надає API для зручного перетворення будь-яких кодувань символів.

Як було зазначено раніше, системний виклик «exec» в POSIX сумісних операційних системах заміняє образ процесу в якому він викликається новим, але в даному випадку нам необхідно створити новий процес. Для цього скористаємося системним викликом «fork», який створює точну копію поточного процесу. В скопійованому процесі викличемо «exec», а в оригінальному дочекаємося завершення виконання копії за допомогою «waitpid».

```
__pid_t e_pid = fork();
if (e_pid == -1) {
  err("Can't fork process for editor", __LINE__ - 1);
  return 1;
}

if (!e_pid && execl("build/editor", "build/editor", (char *)NULL)) {
  err("Can't launch build/editor", __LINE__ - 1);
  return 2;
}

waitpid(e_pid, 0, 0);
```

Повторимо те саме для другої програми.

```
__pid_t a_pid = fork();
if (a_pid == -1) {
  err("Can't fork process for analizer", __LINE__ - 1);
  return 3;
}

if (!a_pid && execl("build/analizer", "build/analizer", s, (char
*)NULL)) {
  err("Can't launch build/analizer", __LINE__ - 1);
  return 4;
}

waitpid(a_pid, 0, 0);
```

7.2.1.4 Тестування

Скомпілюємо всі 3 програми та запустимо головну. Перед нами відкриється редактор у вказаній директорії, створимо в ньому 4 файли з різним кодуванням.

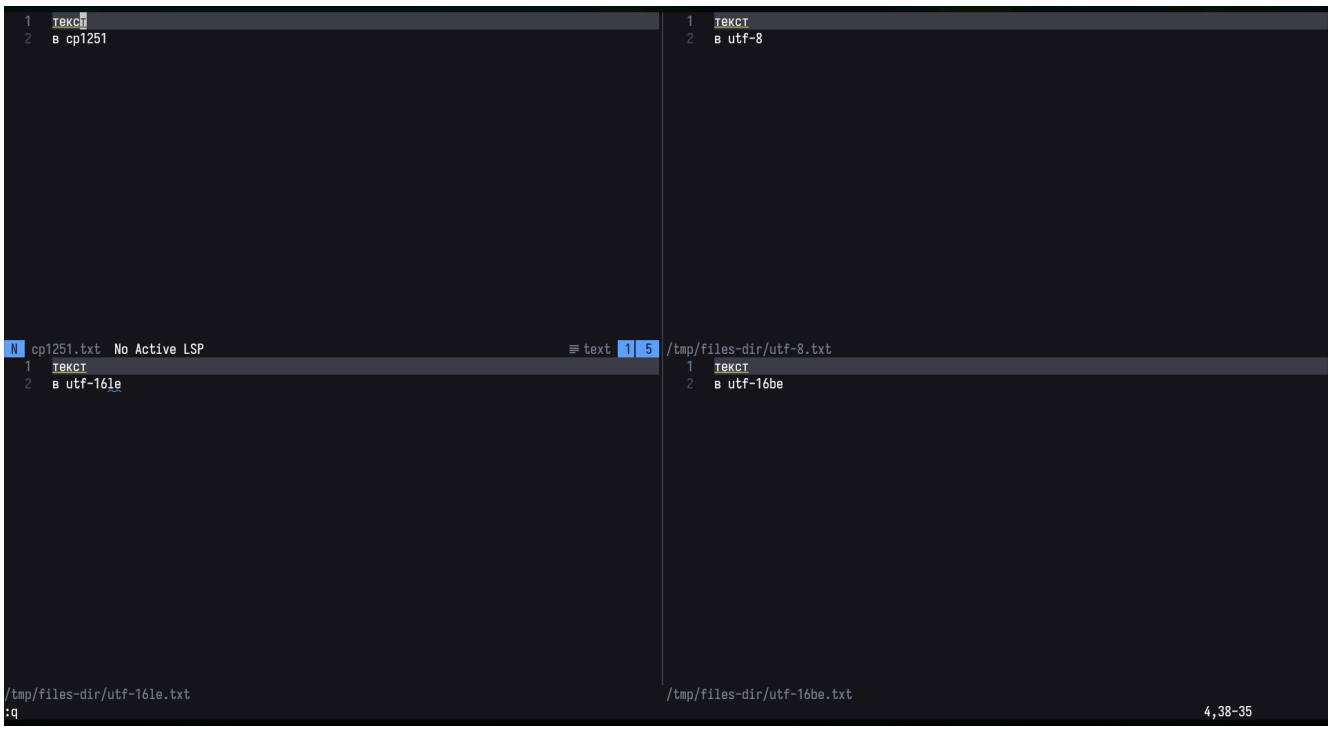

Рисунок 7.1 – Файли з різним кодуванням

Після закриття редактору можемо побачити наступний результат аналізу файлів.

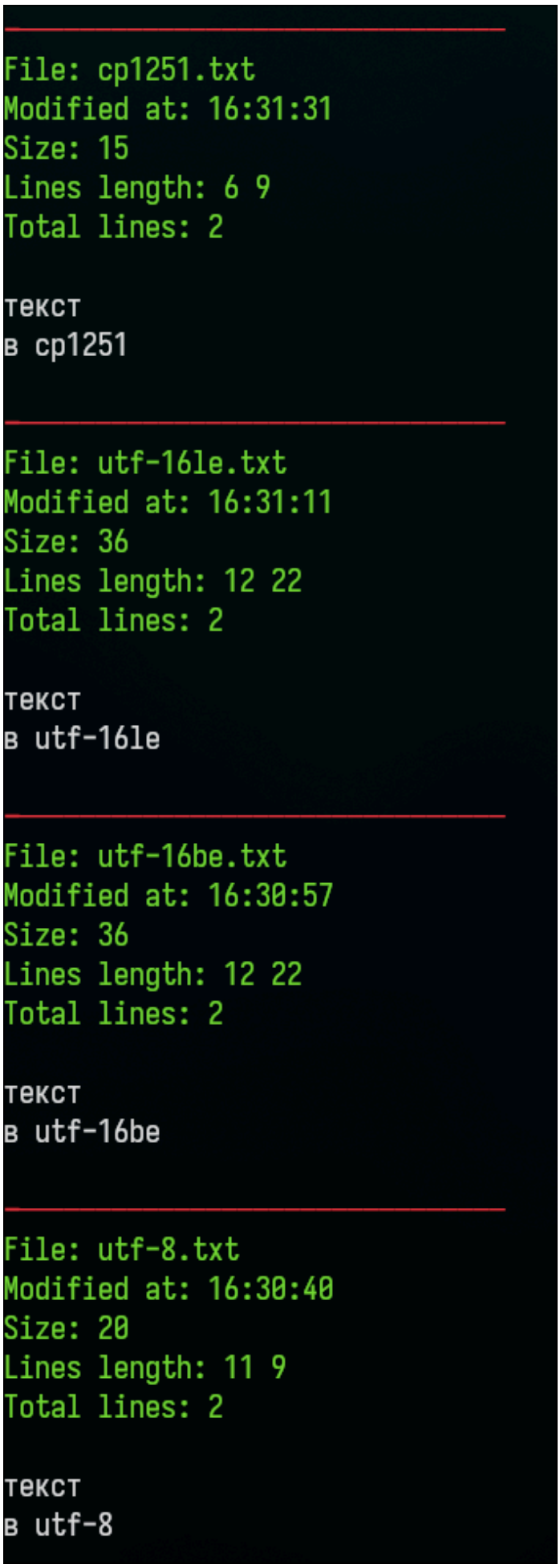

Рисунок 7.2 – Результат аналізу файлів

7.2.2 Завдання 2 – аналіз поведінки потоків.

Метою цього завдання є дослідження поведінки потоків.

Завдання полягає у створенні 10 потоків, та спостереженні за ними.

В POSIX для керування потоками використовується «pthreads».

Створимо 10 потоків, в кожному з них виведемо в консоль повідомлення про початок виконання, виконаємо затратну по часу операцію, та виведемо повідомлення про завершення виконання.

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

#define NUM_THREADS 10

void *thread_function(void *thread_id) {
  long tid = (long)thread_id;
  printf("Begin\t%ld\n", tid);
  for (int i = 0; i < 100000; ++i)
    ;
  printf("End\t%ld\n", tid);
  pthread_exit(NULL);
}

int main() {
  pthread_t threads[NUM_THREADS];
  long t;

  for (t = 0; t < NUM_THREADS; t++)
    pthread_create(&threads[t], NULL, thread_function, (void *)t);

  void *status;
  pthread_join(threads[0], &status);

  printf("Main thread completed execution\n");

  return 0;
}
```

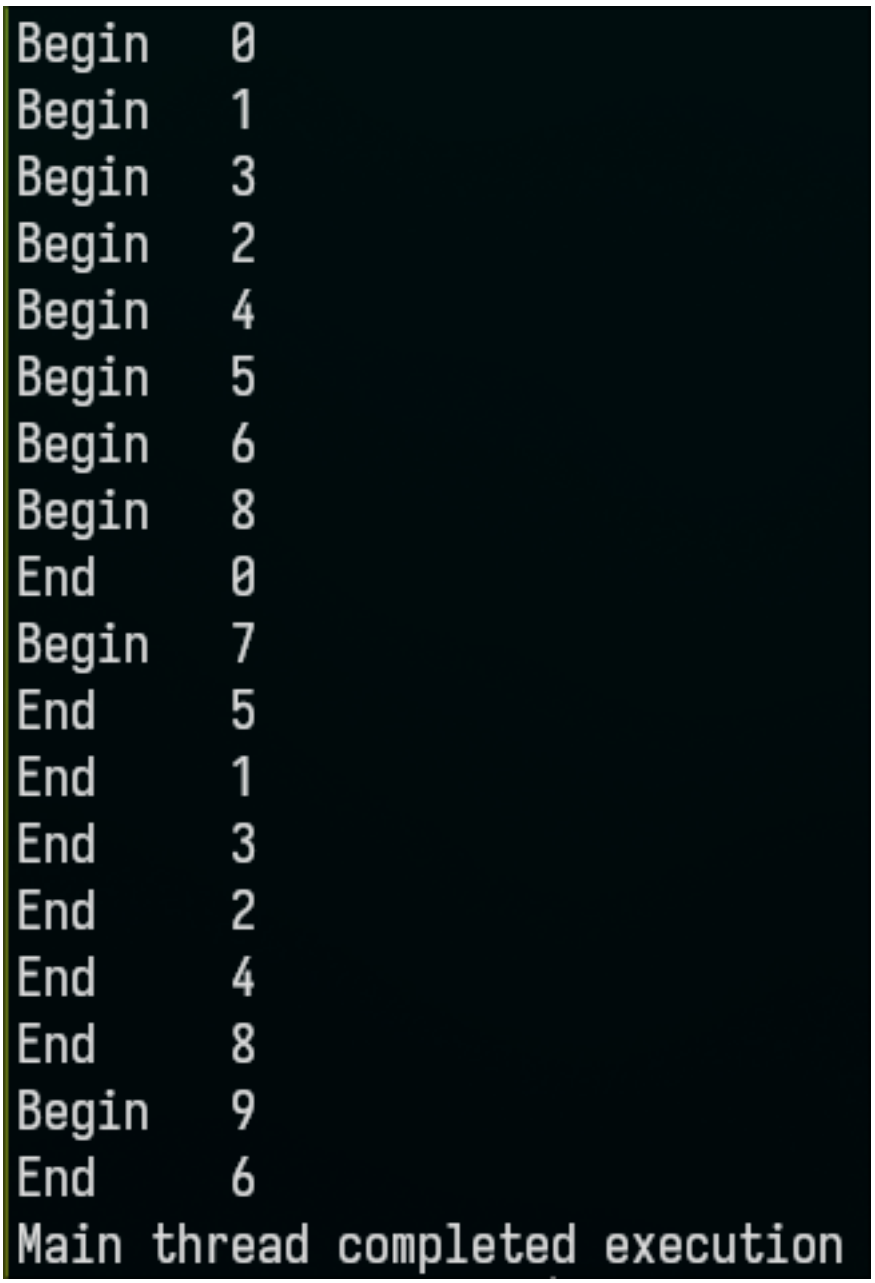

Рисунок 7.3 – Результат виконання

Можемо побачити, що потоки починають виконання в хаотичному порядку. Це пояснюється тим, що операційна система передає потоку виконання незважаючи на час, коли потік було створено. Також можемо побачити, що серед виводу нема повідомлень про завершення потоків під номерами 7, 8 та 9. Оскільки головна програма чекає на завершення лише одного потоку, всі потоки, що не встигли завершити своє виконання до виходу першого автоматично завершуються операційною системою. Щоб це виправити необхідно або від’єднувати потоки від головної програми, або чекати на завершення всіх потоків, а не тільки першого.

7.2.3 Завдання 3 – порівняння швидкодії послідовних та паралельних обчислень.

Метою цього завдання є порівняння швидкодії послідовних та паралельних обчислень.

Завдання полягає у створенні програм для паралельного та послідовного множення матриць.

Множення матриць це дуже ресурсо-затратна операція, настільки затратна, що для її виконання використовується спеціалізоване обладнання – графічні прискорювачі, які можуть виконувати тисячі паралельних операцій.

Створимо функції для послідовного та паралельного множення матриць.

```
void mul_seq(double **a, double **b, double **r, int size) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      r[i][j] = 0;
      for (int k = 0; k < size; k++)
        r[i][j] += a[i][k] * b[k][j];
    }
  }
}

void *mul_par_worker(void *args) {
  ThreadArgs *t_args = (ThreadArgs *)args;
  double **a = t_args->a;
  double **b = t_args->b;
  double **r = t_args->r;
  int start_row = t_args->start_row;
  int end_row = t_args->end_row;
  int size = t_args->size;

  for (int i = start_row; i < end_row; i++) {
    for (int j = 0; j < size; j++) {
      r[i][j] = 0;
      for (int k = 0; k < size; k++)
        r[i][j] += a[i][k] * b[k][j];
    }
  }

  pthread_exit(NULL);
}

void mul_par(double **a, double **b, double **r, int size, int n) {
  pthread_t threads[n];
  ThreadArgs args[n];
  int rows_per_thread = size / n;
  int rem_rows = size % n;
  int curr_row = 0;

  for (int i = 0; i < n; i++) {
```

```
    args[i].a = a;
    args[i].b = b;
    args[i].r = r;
    args[i].start_row = curr_row;
    args[i].end_row = curr_row + rows_per_thread + (i < rem_rows ?
1 : 0);
    args[i].size = size;

    if (pthread_create(&threads[i], NULL, mul_par_worker, &args[i]) !
= 0) {
      perror("Can't creat thread");
      exit(EXIT_FAILURE);
    }
    curr_row = args[i].end_row;
  }

  for (int i = 0; i < n; i++) {
    if (pthread_join(threads[i], NULL) != 0) {
      perror("Can't join thread");
      exit(EXIT_FAILURE);
    }
  }
}
```

Перевіримо швидкодію обох функцій, проведемо паралельні обчислення із використанням 4 потоків.

```
performing sequentional multiplication...
time: 0.923881 seconds

performing parrallel multiplication (4 threads)...
time: 0.218064 seconds
```

Рисунок 7.4 – Результати тестів

Можемо побачити, що паралельне обчислення із використанням 4 потоків швидше а послідовне приблизно в 4.4 рази.

Додаткове збільшення кількості потоків не змінює час виконання в кращу сторону, а завелика кількість потоків навіть може погіршити результати. Використовувати більше потоків ніж одночасно підтримує процесор нема сенсу, адже в такому випадку одному потоку доведеться чекати поки виконується інший, і в такому разі час затрачений на створення потоку та перемикання контексту процесора може навіть перевищити час, що було зекономлено за допомогою паралелізації обчислень.

7.3 Висновки

Під час даної лабораторної роботи ми навчились використовувати процеси та потоки при виконанні програм.