

Завдання 1

Щоб реалізувати функцію для виведення інформації про процес із можливістю контролю виводу, організуємо її як багатофайловий проєкт, що складається з основного файлу для запуску програми (main.c), файлу з функцією виведення інформації про процес (process_info.c) і заголовкового файлу (process_info.h) для оголошення функцій.

```
mint@asus-mint:~/git/op-course/lab4/task1$ ./bin/process_info 3
Process ID (PID): 20466
Parent Process ID (PPID): 19658
mint@asus-mint:~/git/op-course/lab4/task1$ ./bin/process_info 5
Process ID (PID): 20467
User ID (UID): 1000
mint@asus-mint:~/git/op-course/lab4/task1$ ./bin/process_info 15
Process ID (PID): 20471
Parent Process ID (PPID): 19658
User ID (UID): 1000
Group ID (GID): 1000
mint@asus-mint:~/git/op-course/lab4/task1$
```

Завдання 2

Для демонстрації непередбачуваності перемикання процесів створимо програму з двома процесами — батьківським і нащадком. Кожен з них у заданий проміжок часу буде виконувати цикл, в якому збільшуватиме лічильник. В результаті ми побачимо, скільки разів кожен процес зміг виконати тіло циклу, що покаже "непередбачуваність" через перемикання між процесами.

```
mint@asus-mint:~/git/op-course/lab4/task2$ ./bin/process_counter 3
Process 21319: Counter reached 1178881913
Process 21320: Counter reached 1172008590
mint@asus-mint:~/git/op-course/lab4/task2$ ./bin/process_counter 10
Process 21326: Counter reached 3705668816
Process 21327: Counter reached 3715790782
```

Завдання 3

Основний процес після створення всіх нащадків виводить список працюючих процесів за допомогою команди ps та пропонує користувачу вибір: завершити процеси або залишити їх у стані виконання для подальшої перевірки.

Завдання 5

Головний файл main.c — реалізовує виклик нашої кастомної функції my_system().

Файл реалізації mysystem.c — містить реалізацію функції my_system.

Заголовковий файл mysystem.h — описує функцію my_system, щоб її можна було використовувати у різних файлах.

```
mint@asus-mint:~/git/op-course/lab4/task5$ ./bin/my_program
total 28
drwxrwxr-x 2 mint mint 4096 Oct 25 11:55 bin
drwxrwxr-x 2 mint mint 4096 Oct 25 11:55 build
drwxrwxr-x 2 mint mint 4096 Oct 25 11:55 include
-rw-r--r-- 1 mint mint  475 Oct 25 11:55 Makefile
drwxrwxr-x 2 mint mint 4096 Oct 25 11:54 src
Command exited with status: 0
mint@asus-mint:~/git/op-course/lab4/task5$
```

Завдання 6

create_zombie(): функція створює зомбі-процес шляхом форкування і негайного завершення дочірнього процесу, не очікуючи його завершення в батьківському процесі.

Команда ps: використовується для демонстрації зомбі-процесу у списку.

Команда wait(): завершує зомбі-процес, видаляючи його зі списку процесів.

```
mint@asus-mint:~/git/op-course/lab4/task6$ ./bin/zombie_app
Parent process, PID: 26671. Checking for zombie process:
Child process (Zombie) PID: 26672
  PID    PPID S CMD
 26644   26623 S bash
 26671   26644 S ./bin/zombie_app
 26672   26671 Z [zombie_app] <defunct>
 26673   26671 S sh -c ps -o pid,ppid,state,cmd
 26674   26673 R ps -o pid,ppid,state,cmd

After calling wait():
  PID    PPID S CMD
 26644   26623 S bash
 26671   26644 S ./bin/zombie_app
 26675   26671 S sh -c ps -o pid,ppid,state,cmd
 26676   26675 R ps -o pid,ppid,state,cmd
mint@asus-mint:~/git/op-course/lab4/task6$
```

Завдання 7

Основна проблема, з якою стикнувся при виконанні цього завдання, полягала в тому, що я не міг нормально повернути кількість влучань у коло. Це відбувалось через те, що в exit()

дочірнього процесу значення `hits` передається як код завершення, але код завершення має обмеження в діапазоні `[0-255]`. Таким чином, коли `hits` перевищує 255, результат передається некоректно, і тому батьківський процес отримує значення, що завжди менше або дорівнює 255.

Використання `pipe`. Створимо `pipe`, який дозволяє дочірньому процесу передати значення `hits` батьківському процесу, уникаючи обмеження `exit()` на передачу чисел більше за 255.

У `pipe` масив `pipefd` має два елементи для позначення двох кінців каналу зв'язку:

1. `pipefd[0]` – кінець для читання з `pipe`.
2. `pipefd[1]` – кінець для запису в `pipe`.

Відповідно, два кінці потрібні, щоб розділити потоки: один процес (у нашому випадку дочірній) записує дані, а інший (батьківський) читає. Ця конструкція створює односторонній канал, де кожен кінець виконує свою функцію.

Без обох кінців масиву `pipefd` `pipe` не зможе виконувати роль каналу для передачі даних, адже для правильної роботи `pipe` має знати, з якого місця дані записувати, а з якого – читати.

```
mint@asus-mint:~/git/op-course/lab4/task7$ ./bin/monte_carlo_app 3000000 2
After 3000000 throws area is 12.56579733, error is 0.00057328
mint@asus-mint:~/git/op-course/lab4/task7$ ./bin/monte_carlo_app 3000000 2
After 3000000 throws area is 12.56554133, error is 0.00082928
mint@asus-mint:~/git/op-course/lab4/task7$ ./bin/monte_carlo_app 3000000 2
After 3000000 throws area is 12.57370667, error is 0.00733605
mint@asus-mint:~/git/op-course/lab4/task7$ ./bin/monte_carlo_app 3000000 2
After 3000000 throws area is 12.56262400, error is 0.00374661
mint@asus-mint:~/git/op-course/lab4/task7$
```

Висновок

У результаті виконання цієї роботи ми на практиці дослідили основні аспекти роботи з процесами, міжпроцесною взаємодією, каналами зв'язку та методами обчислення в паралельних середовищах. Починаючи від створення базових програм, які використовують функції `fork`, `exec` та `wait`, ми поступово вивчили підходи до передачі даних між батьківським і дочірніми процесами за допомогою аргументів командного рядка, кодів завершення та каналів (`pipes`).

Одним із практичних прикладів стала реалізація обчислення площі круга за методом Монте-Карло, що дозволило нам оцінити принципи створення багатофайлових проєктів, управління кількома процесами та відлагодження їхньої взаємодії. Також було важливо розібратися з методами генерації випадкових чисел для статистичних методів, передачі результатів обчислень між процесами, а також спостереженням за поведінкою процесів за допомогою системних команд.