**Міністерство Освіти І НАУКИ України**

**Національний університет "Львівська політехніка"**

Інститут **ІКНІ**

Кафедра **ПЗ**

**ЗВІТ**

До лабораторної роботи № 4

**На тему:** *“Створення та керування процесами засобами API в операційній системі LINUX ”*

**З дисципліни:** *“Операційні системи”*

**Лектор:**

Старший викладач ПЗ

Грицай О.Д.

**Виконав:**

ст. гр. ПЗ-22

Солтисюк Д.А.

**Прийняв:**

Старший викладач ПЗ

Грицай О.Д.

« \_\_\_\_ » \_\_\_\_\_\_\_\_ 2022 р.

∑= \_\_\_\_

Львів – 2022

**Тема роботи**: створення та керування процесами засобами API в операційній системі Linux.

**Мета роботи**: ознайомитися з багатопоточністю в ОС Linux. Навчитися працювати з процесами, y OC Linux.

**Теоретичні відомості**

Процеси в ОС Linux створюються з допомогою системного виклику fork(). Цей виклик створює точну копію батьківського процесу. Після виконання fork() усі ресурси дочірнього процесу - це копія ресурсів батька. Копіювати процес з усіма виділеними сторінками пам'яті - справа дорога, тому в ядрі Linux використовується технологія Copy-On-Write. Всі сторінки пам'яті батька позначаються як read-only і стають доступні і батькові, і дитині. Як тільки один з процесів змінює дані на певній сторінці, ця сторінка не змінюється, а копіюється і змінюється вже копія. Оригінал при цьому «відв'язується» від даного процесу. Як тільки read-only оригінал залишається «прив'язаним» до одного процесу, сторінці знову призначається статус read-write.

Результат виклику fork() повертається і в батьківський і в дочірній процеси, які починають виконувати однакові інструкції. Відмінність між батьківським і дочірнім процесом полягає лише у :

• Дочірньому процесу присвоюється унікальний PID

• Ідентифікатори батьківського процесу PPID для цих процесів різні

• Дочірній процес вільний від сигналів, що очікують

Значення, що повертає fork() для батьківського це PID дочірнього, а для дочірнього 0.

**ЗАВДАННЯ**

1. Виконати в окремому процесі табулювання функцій.
2. Реалізувати табулювання функцій у 2-ох, 4-ох, 8-ох процесах. Виміряти час роботи процесів. Порівняти результати роботи в одному і в багатьох процесах.
3. Реалізувати можливість зміни пріоритету виконання процесу.
4. Реалізувати можливість зупинки і відновлення роботи процесу
5. Реалізувати можливість вбиття процесу.
6. Порівняти результати виконання програми під ОС Windows та Linux.
7. Результати роботи відобразити у звіті.

**Варіант 7:**Табулювати функцію ln x, задану розкладом в ряд Тейлора, в області її визначення на відрізку від А до В (кількість кроків не менше 100 000 –задається користувачем).

**Хід виконання роботи**

Спочатку створю підпрограму, яка рахуватиме ряди Тейлора на заданому проміжку та виводитиме PID / час виконання після завершення обрахунків:

#include <unistd.h>

#include <chrono>

#include <cmath>

#include <iostream>

#include <string>

void tabulate\_lnx(double a, double b, double step, double iter\_count) {

pid\_t pid = getpid();

**for** (double n = a; n <= b; n += step) {

double num, mul, cal, sum = 0;

num = (n - 1) / (n + 1);

**for** (int i = 1; i <= iter\_count; i++) {

mul = (2 \* i) - 1;

cal = pow(num, mul);

cal = cal / mul;

sum = sum + cal;

}

sum = 2 \* sum;

std::cout << "ln(" << n << ")=" << sum << std::endl;

std::cout << "PID: " << pid << std::endl;

}

}

int main(int argc, char\*\* argv) {

**if** (argc != 5) **return** 1;

double a = atof(argv[1]);

double b = atof(argv[2]);

double step = atof(argv[3]);

int iter\_count = atoi(argv[4]);

std::cout << "a: " << a << " b " << b << " step " << step << " "

<< " iter count" << iter\_count << std::endl;

const auto begin = std::chrono::high\_resolution\_clock::now();

tabulate\_lnx(a, b, step, iter\_count);

const auto time = std::chrono::high\_resolution\_clock::now() - begin;

std::cout << "Duration: "

<< std::chrono::duration<double, std::milli>(time).count() << " ms"

<< std::endl;

getc(stdin);

}

Код програми:

#include <signal.h>

#include <unistd.h>

#include <chrono>

#include <iostream>

int ask\_for\_pid() {

int pid;

std::cout << "Enter PID of the process:" << std::endl;

std::cin >> pid;

**return** pid;

}

int main() {

pid\_t proc[8];

int status[8];

double A, B, step;

int precision, countProc;

std::cout << "Please, enter A (lower bound): " << std::endl;

std::cin >> A;

std::cout << "Please, enter B (upper bound): " << std::endl;

std::cin >> B;

std::cout << "Please, enter step for tabulation for each process: "

<< std::endl;

std::cin >> step;

std::cout << "Please, enter iteration count for each process (precision): "

<< std::endl;

std::cin >> precision;

std::cout << "Please, enter the number of processes: " << std::endl;

std::cin >> countProc;

double rangePerProcess = (B - A) / countProc;

*// creation of processes*

**for** (int i = 0; i < countProc; i++) {

proc[i] = fork();

**if** (proc[i] == -1) {

std::cout << "Error! Could not create child process" << std::endl;

} **else if** (proc[i] == 0) {

auto a = rangePerProcess \* i;

auto b = rangePerProcess \* (i + 1);

std::cout << "Launch with args:" << a << " " << b << " " << step << " "

<< precision << std::endl;

auto cmd =

"\"tell application \\\"Terminal\\\" to do script "

"\\\"/Users/dmytro.soltusyuk/Work/labs/"

"2022/operating\_systems/4\_nixapi/program\_7 " +

std::to\_string(a) + " " + std::to\_string(b) + " " +

std::to\_string(step) + " " + std::to\_string(precision) + "\\\"\"";

std::system(("/usr/bin/osascript -e " + cmd).c\_str());

} **else** {

wait(NULL);

}

}

int option;

**while** (1) {

int priority = 0;

std::cout << std::endl << "Please, choose the action: " << std::endl;

std::cout << "1. Change priority" << std::endl

<< "2. Suspend process" << std::endl

<< "3. Resume process" << std::endl

<< "4. Kill process" << std::endl

<< "5. Quit" << std::endl;

std::cin >> option;

int result;

**switch** (option) {

**case** 1:

std::cout << "Enter priority" << std::endl;

std::cin >> priority;

result = setpriority(PRIO\_PROCESS, ask\_for\_pid(), priority);

std::cout << "priority change result: " << result << " errno: " << errno

<< std::endl;

**break**;

**case** 2:

kill(ask\_for\_pid(), SIGSTOP);

**break**;

**case** 3:

kill(ask\_for\_pid(), SIGCONT);

**break**;

**case** 4:

kill(ask\_for\_pid(), SIGKILL);

**break**;

**default**:

**return** 0;

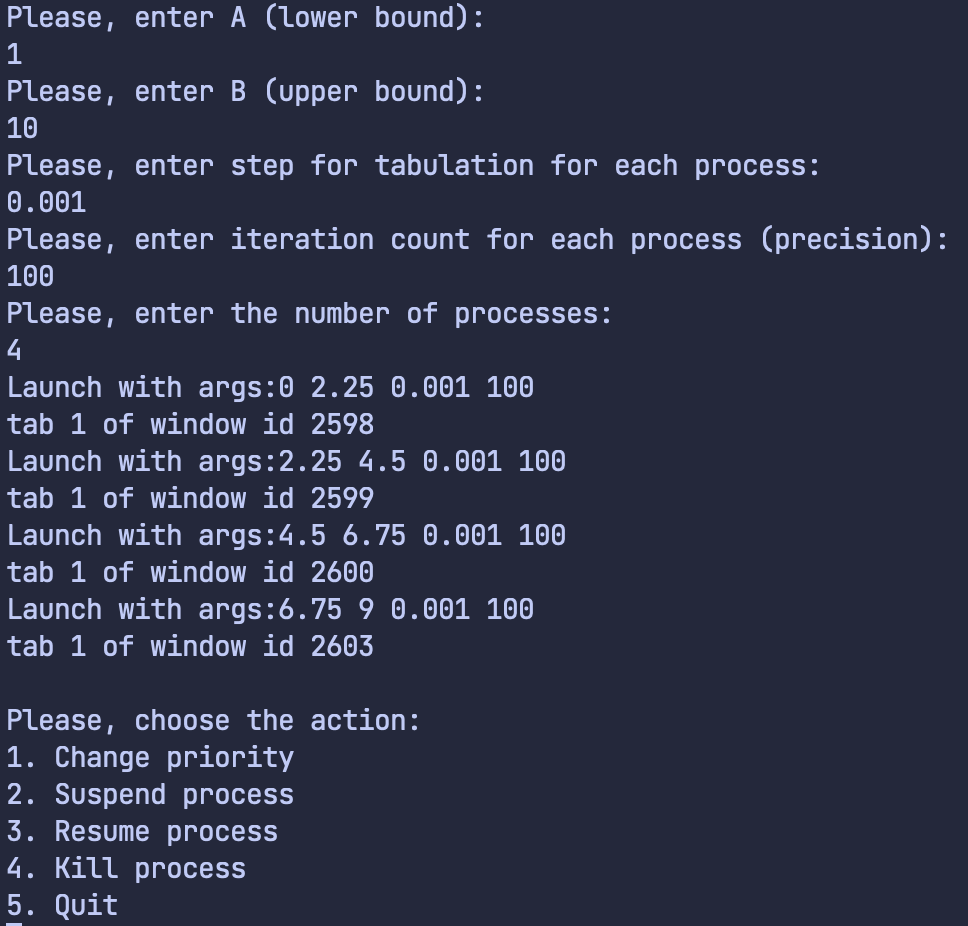
}

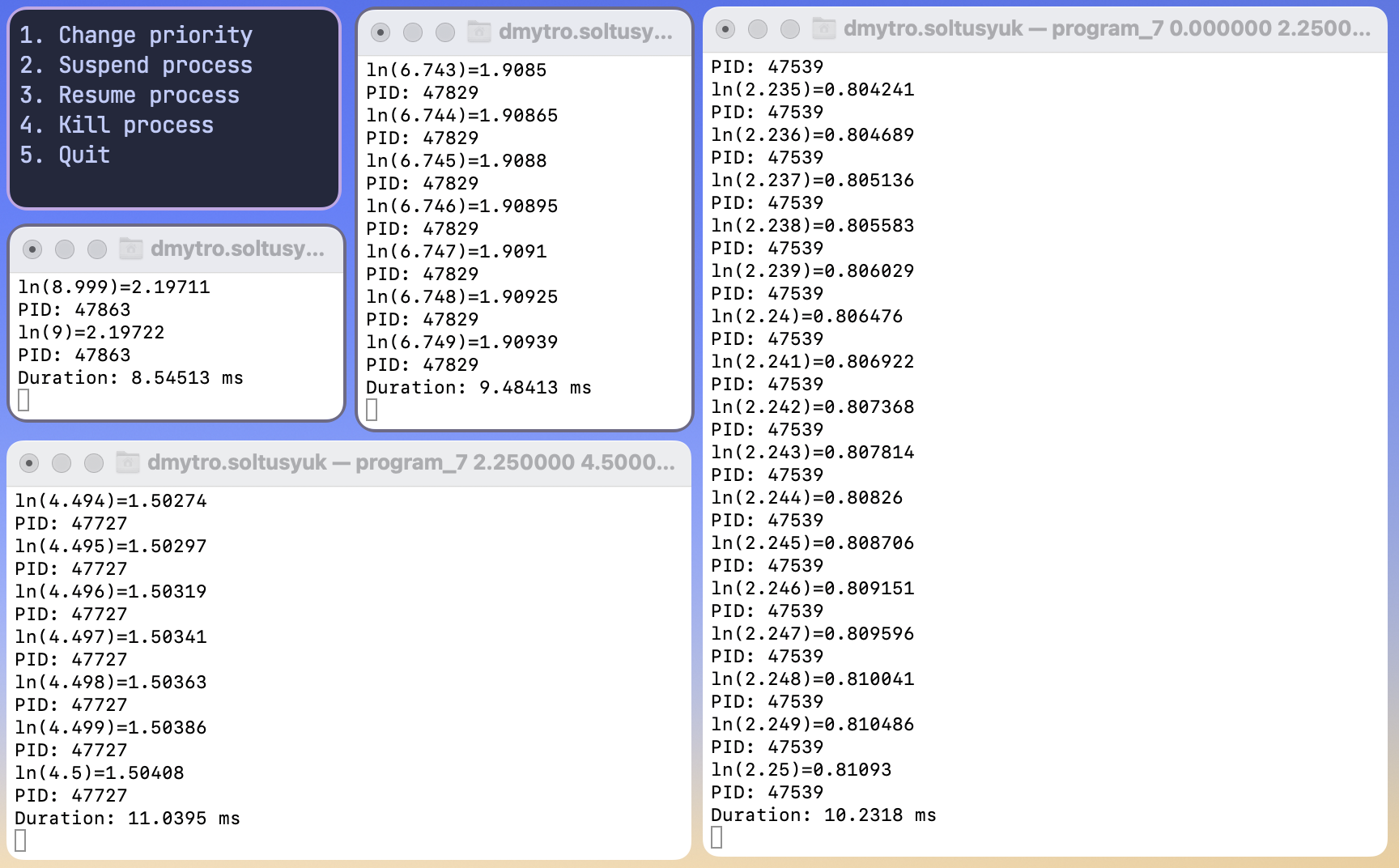
}

**return** 0;

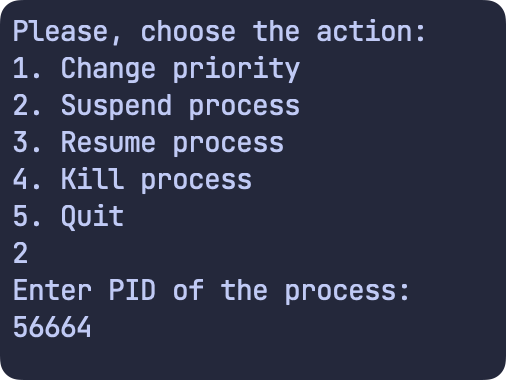
}

**Протокол роботи програми**

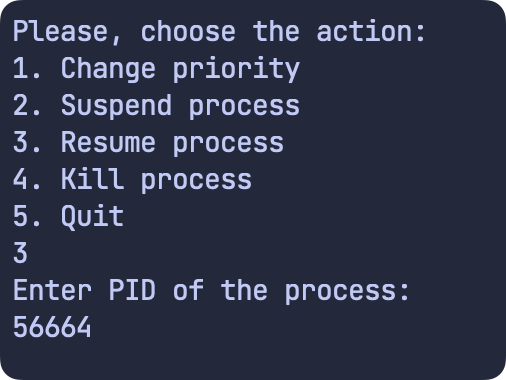
Виконання в 4ьох процесах:

Загальний вигляд роботи програми:

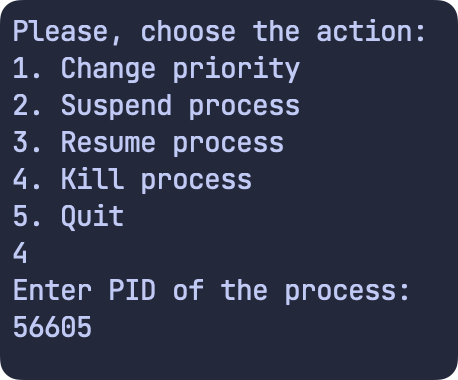
Зупинка процесу



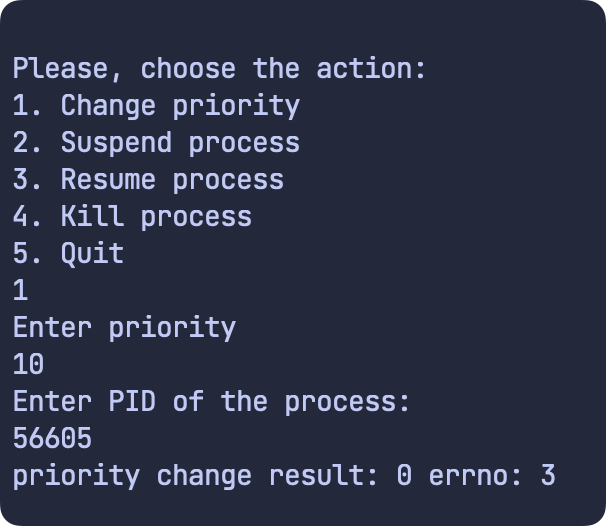
Відновлення процесу



Завершення процесу



Зміна пріоритету



**Висновки**

Виконуючи цю лабораторну роботу, я навчився працювати з Linux API та став краще розуміти, як відбувається взаємодія із процесами на рівні програмного забезпечення на цій операційній системі.