**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»**

Інститут **ІКНІ**

Кафедра **ПЗ**

**ЗВІТ**

До лабораторної роботи №4

На тему: «Програмне створення та керування процесами в операційній системі LINUX»

З дисципліни: «Операційні системи»

**Лектор** : ст.викл каф.ПЗ

Грицай О.Д.

**Виконала:** ст.гр.ПЗ-23

Кохман О.В.

**Прийняла:** ст.викл каф.ПЗ

Грицай О.Д.

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_2022 р.

\_ \_ \_ \_ \_ \_ \_ \_ .

Львів – 2022

Тема: Програмне створення та керування процесами в операційній системі LINUX.

Мета: Ознайомитися з багатопоточністю в ОС Linux. Навчитися працювати з процесами в ОС 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 в:

⦁ 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.Інколи існує необхідність, щоб дочірній процес виконував певну задачу, а батьківський процес лише делегував певні завдання.

Якщо потрібно запустити іншу програму, то необхідно вдатися до системного виклику execve():

int execve (const char \* filename, char \* const argv [], char \* const envp []);

або бібліотечним викликам execl (), execlp (), execle (), execv (), execvp (), execvpe().

П'ята буква визначає вид передачі аргументів: l позначає list, всі параметри передаються як arg1, arg2, ..., NULL, v позначає vector, всі параметри передаються в нуль-термінованому масиві, p позначає path, e позначає environ - у таких викликах останнім аргументом йде нуль-термінований масив нуль-термінованих рядків виду key = value.

**Індивідуальне завдання**

1. Виконати в окремому процесі табулювання функцій ( Можна замінити алгоритмом заданим у лабораторній роботі №3).

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.

****

**Код програми**

Назва файлу: taylor.cpp

#include <stdio.h>

#include <math.h>

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

double A = atof(argv[1]), B = atof(argv[2]), step = atof(argv[3]), accuracy = atof(argv[4]), x, taylor, formula;

int i, sd;

for (x = A; x < B; x += step) {

taylor = 1; i = 2; sd = 1; formula = 0;

while (fabs(sd) > accuracy) {

sd = sd \* (-1) \* x;

taylor += sd \* i;

++i;

}formula = pow(1.0 + x, -2.0);

printf("x = % lf\tformula = % lf\ttaylor = % lf|\tdiv=%lf\n ", x, formula, taylor, fabs(formula - taylor));

}

return 0;

}

Назва файлу:main.cpp

#include <iostream>

#include <unistd.h>

#include <sys/wait.h>

#include <csignal>

#include <sys/resource.h>

#include <chrono>

#include <string>

#define MAX\_NUM\_PROC 8

using namespace std;

int main() {

pid\_t PIDs[MAX\_NUM\_PROC];

int numberOfProcesses;

double MinInter, MaxInter, step, accuracy;

int NumSteps;

cout << "\nEnter number of processes to create: ";

cin >> numberOfProcesses;

cout << "Enter A: ";

cin >> MinInter;

cout << "Enter B: ";

cin >> MaxInter;

cout << "Enter step: ";

cin >> step;

cout << "Enter accuracy: ";

cin >> accuracy;

for (int i = 0; i < numberOfProcesses; ++i) {

string A = to\_string(MinInter);

string B = to\_string(MaxInter);

string Step = to\_string(step);

string Accuracy = to\_string(accuracy);

PIDs[i] = fork();

switch (PIDs[i]) {

case -1:

cout << "Fork error.Something went wrong!";

break;

case 0:

execl("//home//olesia//Documents//taylor", "//home//olesia//Documents//taylor", A.c\_str(), B.c\_str(),

Step.c\_str(), Accuracy.c\_str(), ((char\*)NULL));

break;

default: //parent

{

kill(PIDs[i], SIGSTOP);

cout << "The " << i + 1 << " process PID:" << PIDs[i] << "\nMain process PID: " << getpid() << "\n";

}

}

}

pid\_t wpid;

int status = 0;

while (1) {

int choice;

cout << "\n Measure time (1) Change priority (2) Stop process (3) Resume process (4) Kill process (5)\n";

cin >> choice;

switch (choice) {

case 1: {

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

for (int i = 0; i < numberOfProcesses; ++i) {

kill(PIDs[i], SIGCONT);

}

while ((wpid = wait(&status)) > 0);

auto time = chrono::high\_resolution\_clock::now() - start;

cout << "\nDuration of " << numberOfProcesses << " processes: " << chrono::duration<double, milli>(time).count() \* 0.001 << " s\n";

return 0;

}

case 2: {

int num = 0;

int pr = 0;

cout << "\nEnter process number to change priority: ";

cin >> num;

cout << "\nEnter new number of priority: ";

cin >> pr;

setpriority(PRIO\_PROCESS, PIDs[num - 1], pr);

cout <<

"\nNew priority: " << getpriority(PRIO\_PROCESS, PIDs[num - 1]);

break;

}

case 3: {

int num = 0;

cout << "Enter process number to stop: ";

cin >> num;

if (!kill(PIDs[num - 1], 19)) //сигнал SIGSTOP

{

cout << "\nThe " << num << " process was successfully stopped.\n";

}

else {

cout << "\nError! Something went wrong.\n";

}

break;

}

case 4: {

int num = 0;

cout << "Enter process number to resume: ";

cin >> num;

if (!kill(PIDs[num - 1], 18)) //сигнал SIGCONT

{

cout << "\nThe " << num << " process was successfully resumed.\n";

}

else {

cout << "\nError! Something went wrong.\n";

}

break;

}

case 5: {

int num;

cout << "\nEnter number of process to kill: ";

cin >> num; kill(PIDs[num - 1], SIGKILL);

PIDs[num - 1] = -1;

cout << "\nThe " << num << " process was killed.";

}

}

} wait(NULL);

return 0;

}

**Протокол роботи**

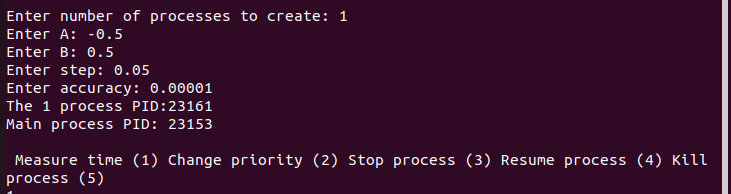
****

Рис. 1 Створення процесу.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кількість | 1 | 2 | 4 | 8 |
| ОС LINUX | 0.00114935 | 0.00117797 | 0.00125077 | 0.00217552 |
| ОС Windows | 13.844 | 12.656 | 16.266 | 30.047 |

Табл. 1 Порівняння часу виконання 1,2,4,8 процесів в OS Linux i OS Windows. З результатів можна зробити висновок, що процеси в лінуксі виконуються набагато швидше , навіть незважаючи на їхню кількість, аніж у віндовсі.

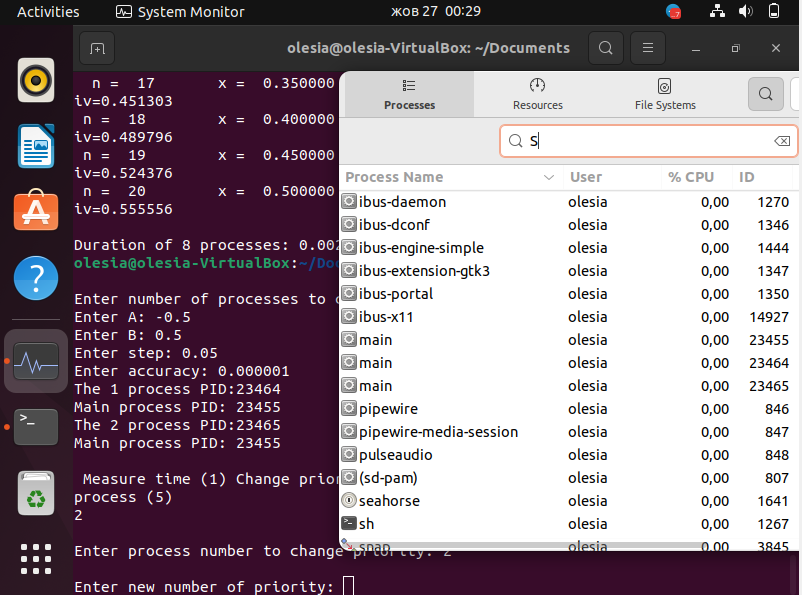
****

Рис. 2 Створення двох процесів у програмі і підтвердження створення в System Monitor.

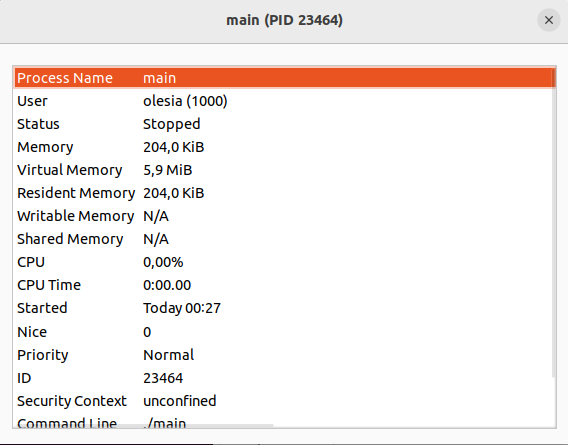
****

Рис. 3 Пріоритет процесу перед зміною пріоритету.

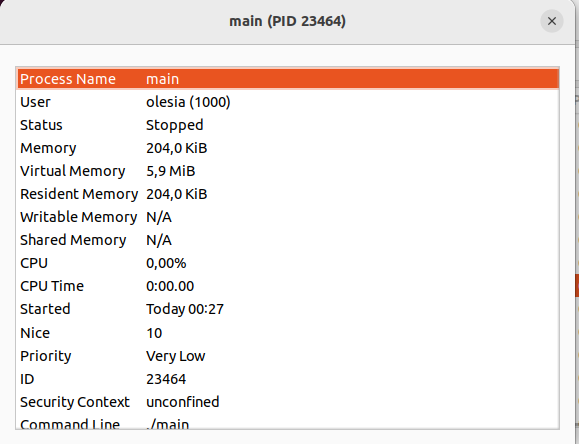
****

Рис. 4 Пріоритет процесу після зміни пріоритету.

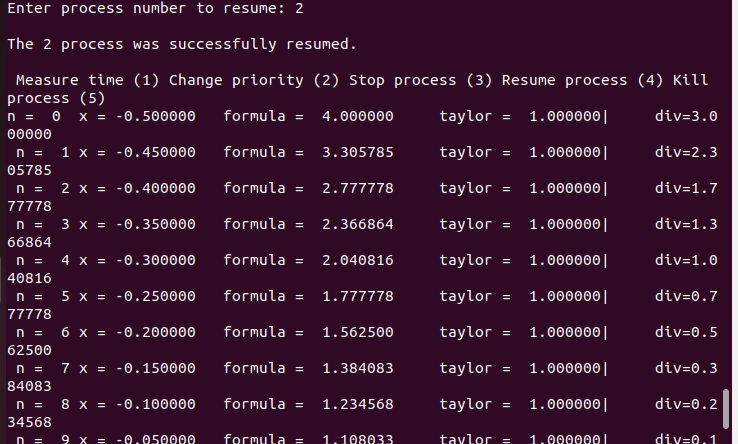
****

Рис. 5 Відновлення процесу.

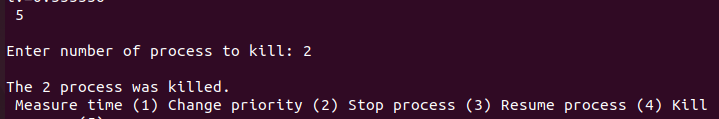
****

Рис. 6 Вбиття процесу.

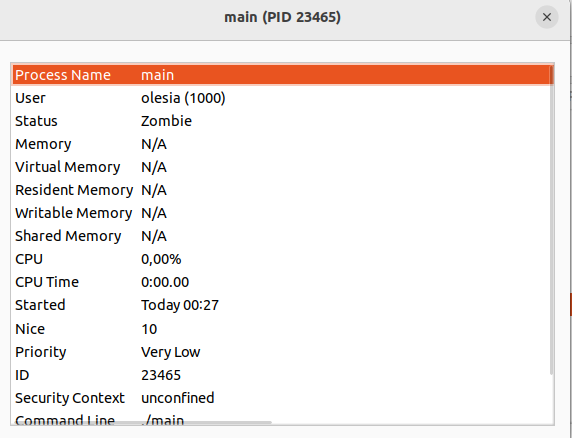
****

Рис. 7 Стан процесу , вбитого перед тим, у System Monitor.

**Висновок**

На цій лабораторній роботі я навчилась працювати з багатопоточність в ОС Linux, а саме: створювати процеси, призупиняти процес, відновлювати процес, зупиняти процес та міняти пріоритет. А також виміряла час виконання 1,2,4,8 процесів і порівняла результати із результатами в ОС Windows за допомогою таблиці.