Міністерство освіти і науки України  
Національний технічний університет України

«Київський політехнічний інститут ім. І. Сікорського»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

Лабораторна робота №7

з дисципліни «Операційна система UNIX»

«Потоки, перенаправлення потоків»

Виконав:

студент 2-го курсу, НН ІАТЕ

групи ТР-23

Ровний Григорій Олександрович

Перевірила:

проф. Левченко Л.О.

КИЇВ 2023

**Мета роботи:**

ознайомитись із системними викликами для управління потоками в ОС Linux, та перенаправлення потоків при роботі файлами і командами.

**Теоретичні відомості**

**Управління потоками.** Процес розбивається на виконувані одиниці - потоки (threads).Потік – набір послідовностей виконуваних команд процесора, які використовуютьзагальний адресний простір процесу, це елемент виконання всередині процесу:віртуальний процесор, стек або статус програми. У порівнянні з процесами взаємодія ісинхронізація потоків вимагає менше часу, оскільки потоки одного процесувиконуються в одному адресному просторі.

Процес містить один або кілька потоків. Якщо процес містить тільки один потік, такий процес називається однопоточними. Це класичні процеси UNIX. Якщо процес містить більше одного потоку, такі процеси називаються багатопоточними. Існує дві основні категорії реалізації потоків: користувацькі потоки та потоки ядра - потоки, що реалізуються через системні виклики і працюють в просторі ядра.

В ОС Linux для потоків реалізований стандарт Р-потоків - POSIX - pthreads. Для написання багатопотокової програми API для роботи з Р-потоками надає біля 100 інтерфейсів. Кожна функція в API забезпечена префіксом pthread\_.

Прототипи функцій роботи з потоками і необхідні типи даних містяться в заголовки <pthread.h>. Ці функції не включені в стандартну бібліотеку мови С, вони знаходяться в бібліотеці libthread. Тому в командному рядку для gсс необхідно додати опцію «- pthread»: *gcc -Wall -Werror -pthread beard.c -o beard.*

Кожнен потік має свій ID потоку. В програмах на С/С++ для ID потоків слід використовувати тип pthread\_t з <sys/types.h>. При роботі з потоками використовуються основні функції: створення потоку; блокування роботи потоку в очікування завершення іншого; дострокове та звичайне завершення роботи потоків;

Створення потоку. Потоки створюються функцією pthread\_create. Ця функція визначена в заголовки <pthread.h>. Перший параметр цієї функції є вказівником на змінну типу pthread\_t, в яку буде записано адресу ідентифікатора створюваного потоку. Другий параметр є вказівником на змінну типу pthread\_attr\_t, використовується для установки атрибутів потоку. Цей об'єкт управляє деталями взаємодії потоку з іншою програмою. Третім параметром функції pthread\_create повинна бути адреса функції потоку – вказівник на функцію потоку. Ця функція відіграє для потоку ту ж роль, що функція main для головної програми. Функція потоку приймає один параметр типу покажчик на void і повертає значення типу вказівник на void. Четвертий параметр функції pthread\_create має тип void\*. Цей параметр може використовуватися для передачі значення як аргумент у функцію потоку. Після виклику pthread\_create функція потоку буде запущена на виконання паралельно з іншими потоками програми. Функція повертає 0 в разі успіху або код помилки – в разі невдачі.

Ідентифікатори потоків (Thread ID) для потоків є аналогами ідентифікаторів процесів (PID). У той час як PID призначаються ядром Linux, TID призначаються бібліотекою Р-потоків. Цей тип представлений pthread\_t, і POSIX не вимагає, щоб він був арифметичним. TID нового потоку визначається за допомогою аргументу thread при успішному виклику pthread\_create().

**Блокування роботи потоку.** Приєднання дозволяє одному з потоків блокуватися в очікуванні завершення іншого.

**Дострокове завершення потоків.** Р-потоки викликають завершення інших потоків через їх скасування. Це забезпечує функція pthread\_cancel(). Успішний виклик pthread\_cancel() надсилає запит на скасування потоку, представленому через ідентифікатор потоку thread. Чи може потік бути скасований і коли, залежить від його стану відміни і типу скасування. Стан відміни потоку може бути доступно або недоступно. Потоки можуть змінювати свій стан через pthread\_setcancelstate():

Тип скасування потоку може бути асинхронним або відкладеним. З асинхронним типом скасування потік може бути убитий в будь-якій точці після отримання команди на скасування. З відкладеним типом потік може бути убитий тільки в спеціальних точках скасування, які є функціями Р- потоків або бібліотеки С і являють собою безпечні моменти

**Завершення роботи потоків.** Завершення роботи потоків схоже на завершення роботи процесів, за винятком, коли потік завершується, інші потоки в процесі продовжують виконуватися. Потоки можуть перериватися:

- якщо потік повертається зі стартової процедури, він переривається;

- якщо потік викликає функцію pthread\_exit(), він завершується;

- якщо потік скасовується іншим потоком через функцію pthread\_cancel(), він завершується; це аналог відправки сигналу SIGKILL через kill(). Найпростіший шлях потоку для завершення самого себе, - це «вихід за границі» своєї початкової процедури. Однак часто потрібно завершити потік десь в глибині стека виклику функції, достатньо далеко від стартової процедури. Для таких випадків в Р- потоках є виклик pthread\_exit(), потоковий еквівалент exit().

**Синхронізація потоків**

Усі потоки виконуються в одному адресному просторі. У зв'язку з цим постає проблема спільного використання загальних змінних, доступу до певного ресурсу, оскільки в один момент часу тільки єдиний потік повинен працювати з певним розділюваним ресурсом. Така задача має назву забезпечення взаємовиключення, а ділянки програмного коду, в яких потоки виконують операції з розділюваними ресурсами, називаються критичними секціями. З іншого боку можлива ситуація, коли потоку для продовження своєї роботи потрібно результат виконання іншого потоку, що потребує синхронізації дій. Для узгодженості взаємодії потоків розроблені засоби синхронізації потоків - м'ютекси (взаємні виключення), семафори і умовні змінні.

М’ютекс дозволяє потокам управляти доступом до даних. При використанні м’ютекса тільки один потік в певний момент часу може заблокувати м’ютекс і отримати доступ до ресурсу (право на використання). По завершенні роботи з ресурсом потік повинен повернути це право, розблокувавши м’ютекс. Якщо будь-який потік звернеться до вже заблокованого м’ютексу, то він буде змушений чекати розблокування м’ютекса потоком, який їм володіє.

Семафор призначений для синхронізації потоків щодо дій та даних. Семафор – це захищена змінна, значення якої можна опитувати і міняти тільки за допомогою спеціальних операцій P і V і операції ініціалізації. Семафор може приймати ціле невід'ємне значення. При виконанні потоком операції P над семафором S значення семафора зменшується на 1 при S> 0 або потік блокується, «чекаючи на семафорі», при S=0. При виконанні операції V(S) відбувається пробудження одного з потоків, які очікують на семафорі S, а якщо таких немає - значення семафора збільшується на 1. При вході в критичну секцію потік повинен виконувати операцію P, а при виході з критичної секції операцію V. Прототипи функцій для маніпуляції з семафора описуються у файлі semaphore.h.

**Поставлене завдання:**

1. Використати вихідні тексти, наведених прикладів для потоків, м’ютексів та семафорів, для створення програм.
2. Пояснити та прокоментувати результати роботи програм та їх особливості.
3. Створити файл з особистими даними (група, ПІБ), використовуючи стандартне введення з клавіатури. Вивести дані на екран. Додати у файл інформацію про ваше хоббі. Створити канал.

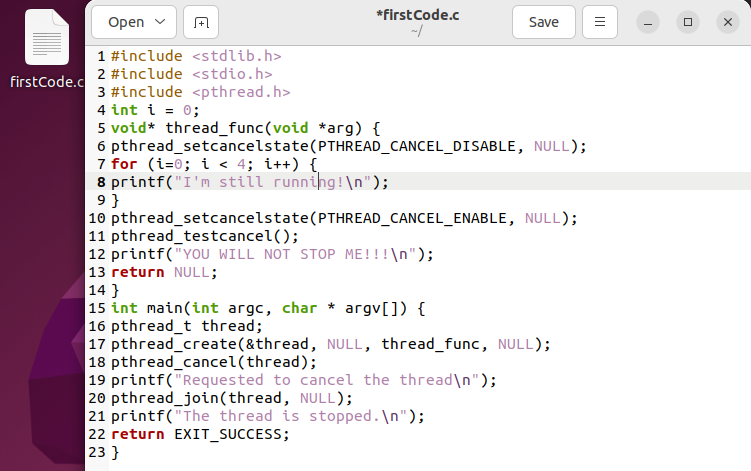
**Результат виконання роботи**

1. Для написання та редагування коду будемо використовувати вбудований текстовий редактор Ubuntu. Для виконання першого коду необхідно створити файл з розширенням .c, для цього використовуємо команду *touch firstCode.c.* Після виконання команди відкриваємо файл та копіюємо до нього код з першого прикладу.

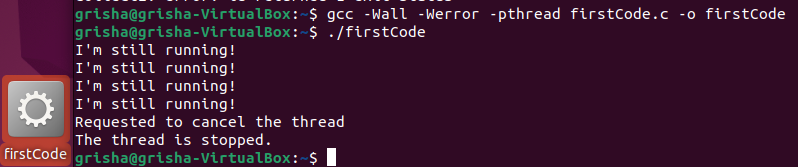


**Код №1 – робота з потоком.** На початку коду ми підключаємо необхідні бібліотеки для роботи з: виводом (stdio.h), управління пам’яттю (stdlib.h) та роботою з потоками (pthreads.h). Далі оголошуємо глобальну змінну i, яка буде використання у якості лічильника у функції thread\_func. У тілі функції thread\_func встановлюється стан відміни потоку на PTHREAD\_CANCEL\_DISABLE, що вказує на те, що потік не може бути відміненим поки виконується деякий код. Далі за допомогою циклу for 4 рази буде виводитись повідомлення «I’m still running!». Потім знову змінюємо стан потоку на PTHREAD\_CANCEL\_ENABLE, що вказує на те, що потік можливо відмінити. Надалі викликається pthread\_testcancel(), що створює точку відміни в межах викликаючого потоку, так що потік, який в іншому випадку виконує код, який не містить точок відміни, буде відповідати на запит відміни. І на кінець виводимо текст «You will not stop me!!!» та повертаємо null.

У функції main, яка є точкою входу, створюємо змінну thread для зберігання ідетнифікатора потоку. Створюємо новий потік pthread\_ctreate, який викликає thread\_func та передає NULL як аргумент. Потім відправляємо запит на відміну потоку (pthread\_cancel) та виводимо відповідне повідомлення. pthread\_join використовується для того щоб потік з основною функцією почекав завершення роботи потоку thread. Після успішного завершення, виводимо повідомлення «The thread is stopped» про зупинку потоку, та повертаємо успішний статус виконання.

****

Зберігаємо код та компілюємо його за допомогою *gcc -Wall -Werror -pthread firstCode.c -o firstCode,* та виконуємо *./firstCode.*



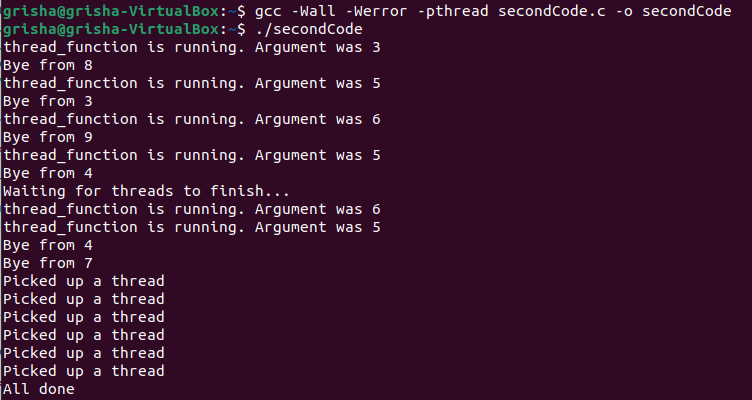
Як бачимо, створений потік виконав вивід тексту чотири рази, після чого виконалося дострокове завершення роботи потоку.

1. **Код №2 – багатопоточні програми.** Окрім стандартних бібліотек, у цьому коді ми використаємо time.h та unistd.h для роботи з часом та стандартними функціями UNIX відповідно. Спочатку оголошуємо константу, яка вказує на кількість потоків, яку необхідно створити. Оголошуємо прототип функції thread\_function(). У головній функції main() створюємо декілька змінних: res – для зберігання результату об’єднання потоків; lots\_of\_threads - виступає в ролі лічильника циклу; a\_thread – масив, який зберігатиме ідентифікатори потоків та thread\_results – для отримання результату роботи потоку при об’єднуванні. Також будемо використовувати srand() для генерації випадкових чисел. Далі за допомогою циклу for створюємо 6 потоків, кожен з яких виконує функцію thread\_function. У випадку, якщо створення не виконується то отримаємо вивід відповідного повідомлення, та викликаємо exit() для завершення роботи. Після чого інший цикл for чекає завершення роботи кожного потоку перед продовженням роботи, при чому порядок завершення роботи потоків не гарантується, тому буде виводитись повідомлення для кожного об’єднаного потоку. Коли все завершилось, отримуємо повідомлення “all done” та робота програми припиняється.

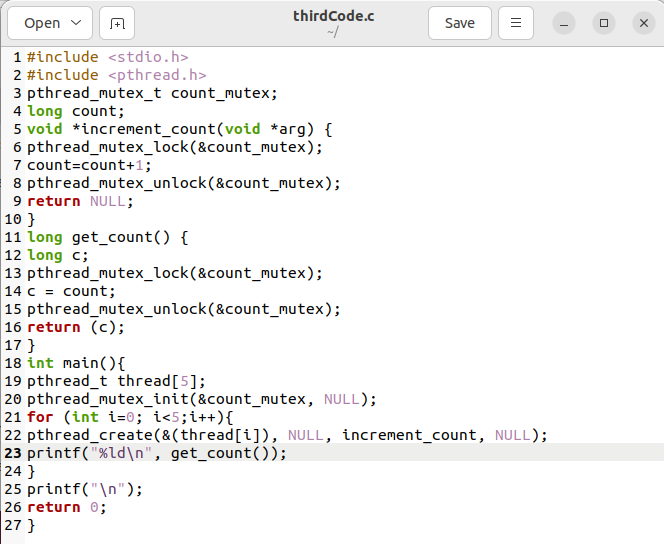




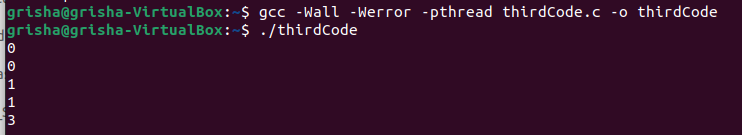
Зберігаємо код та компілюємо його за допомогою *gcc -Wall -Werror -pthread secondCode.c -o secondCode,* та виконуємо *./secondCode.* В результаті виконання бачимо, що було створено 6 потоків, після чого почергово виконувалась зупинка потків, коли потоки були зупинені – робота програми успішно завершилась.



1. **Код №3 – використання м’ютекса.** На початку оголошуємо змінну count\_mutex для створення м’ютексу та count для зміни потоку. Створюємо функцію increment\_count, яка спочатку блокує м’ютекс, що дозволяє одному потоку отримати доступ до ресурсу та уникнути непередбачених результатів через одночасний доступ декількох потоків до змінної, потім відбувається інкрементування змінної count, після чого м’ютекс знову стає розблокованим. Також додаємо функцію get\_count яка допоможе отримати значення змінної count. В main() ми створюємо масив для зберігання ідентифікаторів потоків, ініціалізуємо м’ютекс та викликамо цикл для створення 5 потоків, кожен з яких буде викликати increment\_count та виводити значення count.



Зберігаємо код та компілюємо його за допомогою *gcc -Wall -Werror -pthread thirdCode.c -o thirdCode,* та виконуємо *./thirdCode.* В результаті виконання бачимо, що відбувається інкрементування потоком та вивід змінної до якої застосовується м’ютекс.

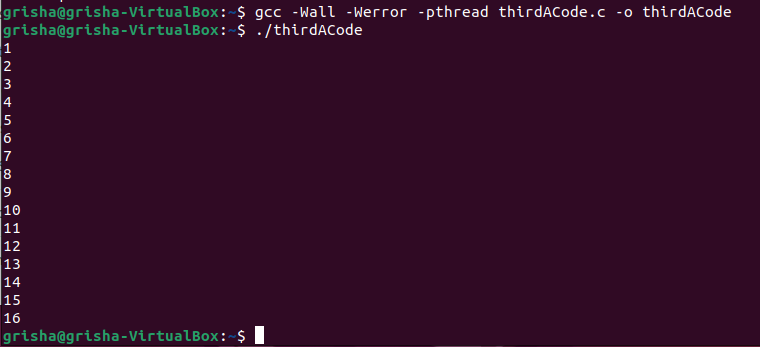


1. **Код №3A – робота з м’ютексом.** Аналогічно до попереднього коду створюємо змінні count\_mutex, count та функцію для інкрементування і виведення. В main оголошуємо масив для зберігання ідентифікаторів потоків, яких цього разу буде 16. Ініціалізуємо м’ютекс та запускаємо цикл. У циклі створюються 16 потоків, кожен з яких буде викликати інкрементацію. У іншому циклі буде виконуватися завершення роботи кожного потоку за допомогою pthread\_join. На кінець за допомогою pthread\_mutex\_destroy викликається знищення м’ютексу.

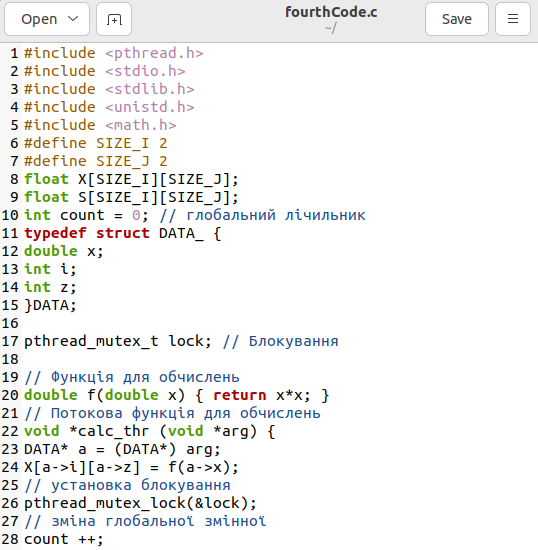


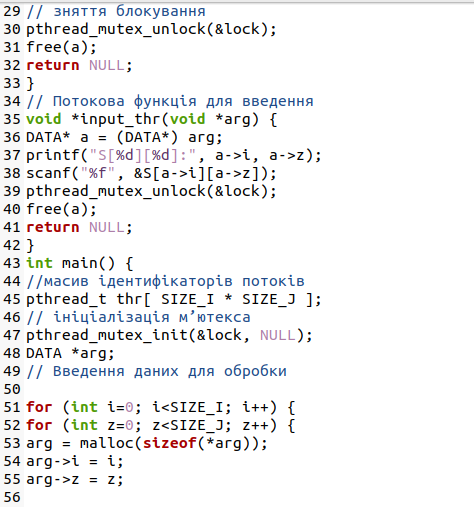


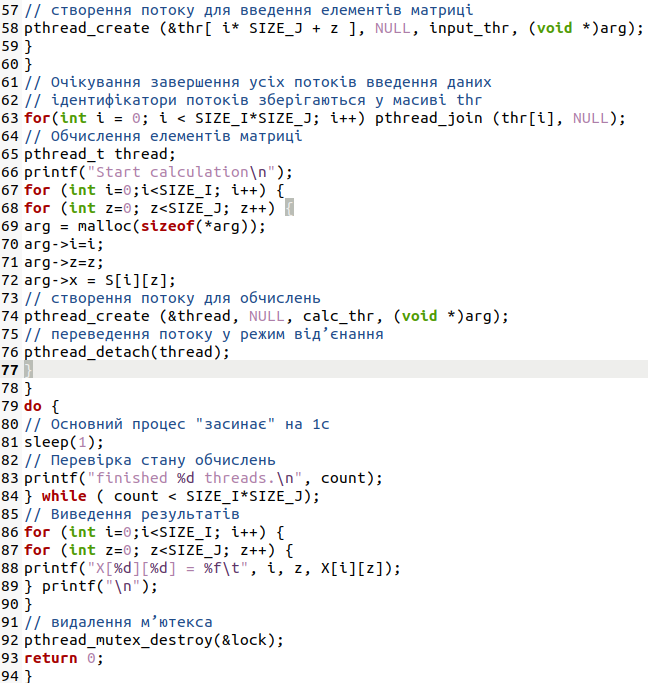
Зберігаємо код та компілюємо його за допомогою *gcc -Wall -Werror -pthread thirdACode.c -o thirdACode,* та виконуємо *./thirdACode.* В результаті виконання бачимо, що відбувається інкрементування потоком та вивід змінної до якої застосовується м’ютекс.



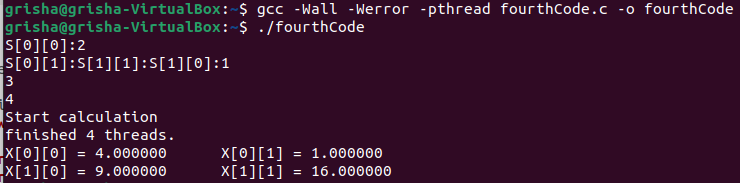
1. **Код №4 – багатопотокові програми з використанням м’ютексів.** В даному коді за допомогою використання м’ютексів відбувається обчислення значень деякої матриці, значення якої ми будемо задавати з клавіатури. На початку коду оголошуємо змінні м’ютекса, та структуру DATA, яка забезпечує передачу даних між основною та потоковою функціями. Потокова функція calc\_thr() обчислює значення функції та зберігає отриманий результат частині матриці; блокує м’ютекс, інкрементує значення змінної count та знімає блокування м’ютекса а також звільняє пам’ять для переданого аргумента. За допомогою input\_thr забезпечується введення значень з клавіатури, зберігання їх у відповідних часнинах матриці та звільнення пам’яті. У функції main() спочатку ініціалізуємо масив, який буде зберігати ідентифікатори потоків. За допомогою циклів забезпечується створення потоків для введення даних, обчислення значень матриці, перемикання режиму роботи потоків, виклик функцій, звільнення пам’яті, вивід результатів та знищення м’ютекса.



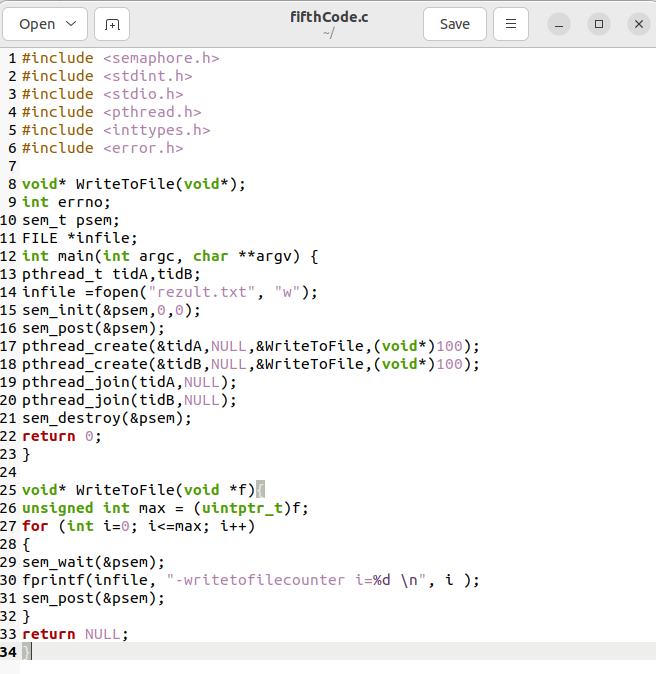




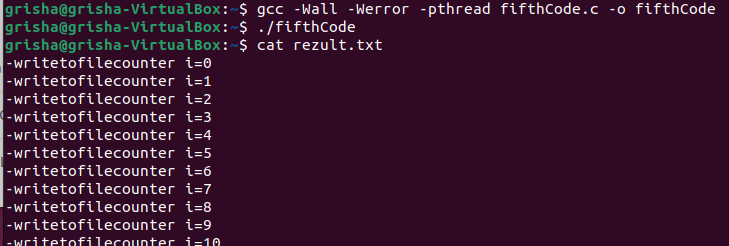
Зберігаємо код та компілюємо його за допомогою *gcc -Wall -Werror -pthread fourthCode.c -o fourthCode,* та виконуємо *./fourthCode.* В результаті виконання бачимо, що спочатку ми створюємо та задаємо числа у двовимірний масив розміром 2х2. Після чого створюються потоки, які виконуватимуть команди функцій для обчислення результату, і в результаті чого ми отримуємо обраховані значення та індекс завершеного потоку.



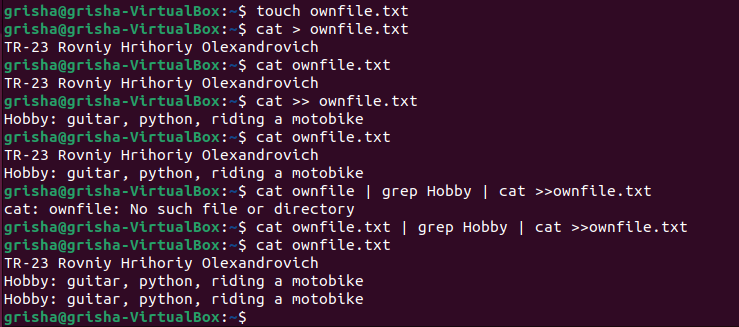
1. **Код №5 – багатопотокові програми з синхронізацією семафона.** Після підключення всіх бібліотек створюємо прототип функції WriteToFile() для запису результату у файл. Оголошуємо змінну semt\_t psem семафору, для управління доступом до файлу. У головній функції створюємо 2 потоки A та B. Далі відкриваємо файл в режим запису. Ініціалізовуємо семафор зі значенням 0, що означає що семафор у заблокованому стані. Викликаємо функцію для збільшення значення семафору, щоб вивести його із заблокованого стану. Після цього запускаємо два потоки з викликом функції запису у файл. Завершуємо роботу обох потоків. Коли запис буде завершено, об’єднуємо потоки, закриваємо файл, та знищуємо семафор.



Зберігаємо код та компілюємо його за допомогою *gcc -Wall -Werror -pthread fifthCode.c -o fifthCode,* та виконуємо *./ fifthCode.* Отриманий результат був записаний у файл result.txt, тому необхідно прочитати його за допомогою cat.



1. Створюємо порожній файл, в який запишемо особисті дані ownfile.txt. Для введення даний у файл застосуємо cat > ownfile.txt. Перевіримо вміст файлу, за допомогою cat. Щоб створити канал для пошуку за значенням «Hobby», використаємо утиліту grep <значення>.



**Висновок:**

В результаті виконання лабораторної роботи ми ознайомилися з системними викликами для управлінням потоками в OC Unix. За допомогою мови програмування C реалізовано декілька прикладів для роботи та управління потоками і багатозадачністю, використання м’ютексів та семафонів; отримано практичні навички зі створення каналів для роботи з файлом.