Міністерство освіти і науки України
НТУУ «КПІ ім. Ігоря Сікорського»
Навчально-науковий інститут атомної та теплової енергетики
Кафедра цифрових технологій в енергетиці

Лабораторна робота №7

з дисципліни «Операційна система UNIX»

Тема «Потоки, перенаправлення потоків»

Варіант №22

Студента 2-го курсу НН ІАТЕ гр. ТР-12
Ковальова Олександра
Перевірила: д.т.н., проф. Левченко Л. О.

КИЇВ 2022

Мета роботи. Набути навичок управління потоками в операційній системі Linux, ознайомитись з системними викликами для управління потоками, навчитись перенаправляти потоки при роботі з файлами та командами. Ознайомитись з поняттям потоків введення та виведення. Освоїти концепцію каналів.

Теоретична частина.

Процес розбивається на виконувані одиниці – потоки (threads) (один і більше потоків). Потік – набір послідовностей виконуваних команд процесора, які використовують загальний адресний простір процесу, це елемент виконання всередині процесу: віртуальний процесор, стек або статус програми. У порівнянні з процесами взаємодія і синхронізація потоків вимагає менше часу, оскільки потоки одного процесу виконуються в одному адресному просторі.

Також, можна сказати, що поток – це контекст виконання процесу [1]. Кожний процес має як мінімум один поток, але деякі можуть мати й декілька. Якщо процес містить тільки один потік, такий процес називається однопоточними. Це класичні процеси UNIX.

В сучасних комп'ютерних системах використовується декілька центральних процесорів та декілька ядер всередині кожного центрального процесора. Такі багатопотокові програми як Apache отримують максимальну користь з мультиядерних систем завдяки тому, що ці додатки можуть працювати з багатьма запитами одночасно.

Якщо процес містить більше одного потоку, такі процеси називаються багатопоточними.

Існує дві основні категорії реалізації потоків: користувацькі потоки – потоки, що реалізуються через спеціальні бібліотеки потоків і працюють в просторі користувача. Потоки ядра – потоки, що реалізуються через системні виклики і працюють в просторі ядра.

В ОС UNIX/Linux для потоків реалізований стандарт P-потоків – POSIX (Portable Operating System Interface) – pthreads ("P" – від POSIX). Для написання багатопотокової програми API для роботи з P-потоками надає біля 100 інтерфейсів. Кожна функція в API забезпечена префіксом pthread_.

Кожний потік має свій ідентифікатор потоку, ID потоку. В програмах на C/C++ 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 завершення потоків;
- завершення роботи потоків.

Потоки створюються функцією pthread_create.

Ідентифікатори потоків (TID – Thread ID) для потоків є аналогами ідентифікаторів процесів (PID). У той час як PID призначаються ядром Linux, TID призначаються лише бібліотекою P-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(). Потік може отримати свій TID при запуску за допомогою функції pthread_self().

Поток можна заблокувати двома методами – за допомогою pthread_join() та pthread_detach(). У першому випадку один потік чекає завершення іншого, у другому потік який не встиг за ходом роботи першого просто переривається.

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

- якщо потік повертається зі стартової процедури, він переривається; це аналог «виходу за границі» в main();
- якщо потік викликає функцію pthread_exit(), він завершується; це аналог виклику exit();
- 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 P потоках є виклик pthread_exit(), потоковий еквівалент exit().

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.

Синхронізація і взаємовиключення забезпечуються за рахунок атомарності виконуваних операцій над м'ютексами і семафорами. Атомарною називають операцію, яка не може бути перервана в ході свого виконання.

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 його 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 семафор», при $S = 0$. При виконанні операції V(S) відбувається пробудження одного з потоків, які очікують на семафор S, а якщо таких немає, то його значення збільшується на 1. Як випливає з вищесказаного, при вході в критичну секцію потік повинен виконувати операцію P, а при виході з критичної секції операцію V.

Стандартні потоки введення і виведення в Linux є одним з найбільш поширених засобів для обміну інформацією процесів, а перенаправлення «>», «>>» і «|» є однією з

найбільш популярних конструкцій командного інтерпретатора. Введення і виведення розподіляється між трьома стандартними потоками:

- stdin – стандартне введення (клавіатура), номер потоку - 0;
- stdout – стандартне виведення (екран), номер потоку - 1;
- stderr –стандартна помилка (виведення помилок на екран), номер потоку – 2.

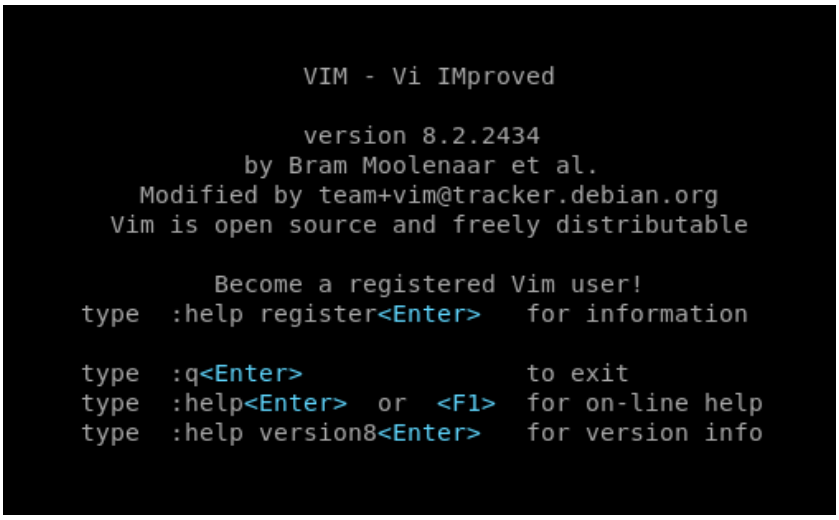
Зі стандартного введення команда може тільки зчитувати дані, а два інших потоки можуть використовуватися тільки для запису. Дані виводяться на екран і зчитуються з клавіатури, так як стандартні потоки за замовчуванням асоційовані з терміналом користувача. Потоки можна підключати до чого завгодно: до файлів, програм і навіть пристроїв.

Канали використовуються для перенаправлення потоку з однієї програми в іншу. Особливим варіантом перенаправлення виведення є організація програмного каналу (іноді називають «пайпою» або конвеєром, оператор «|»). Для цього дві або декілька команд, таких, як виведення попередньої слугує введенням для наступної та розділяються символом вертикальної риски – «|». При цьому стандартний вихідний потік команди, розташованої ліворуч від символу «|», направляється на стандартне введення програми, розташованої праворуч від символу «|».

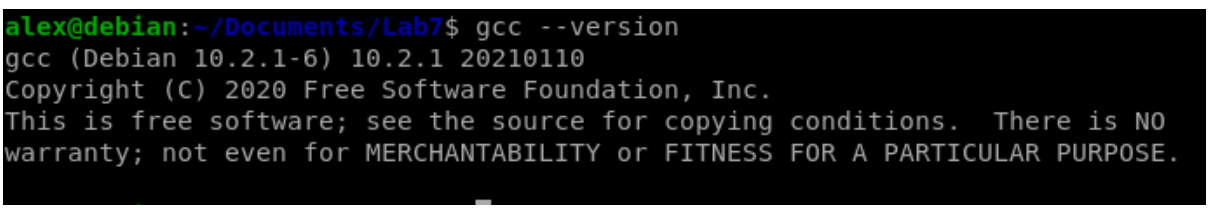
Програмні канали використовуються для того, щоб скомбінувати кілька маленьких програм, кожна з яких виконує тільки певні перетворення над своїм вхідним потоком, для створення узагальненої команди, результатом якої буде якесь більш складне перетворення.

Хід роботи

При написанні та редагуванні коду використовувався текстовий редактор Vim:



Компіляція програмного коду відбувалася за допомогою компілятора gcc (GNU Compiler Collection):



Програмний код першого прикладу:

alex@debian: ~/Documents/Lab7 alex@debian: ~/Documents/Lab7

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

void* thread_func(void *arg) {
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    for (int i = 0; i < 4; i++) {
        printf ("I'm still running!\n");
    }
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_testcancel();
    printf("YOU WILL NOT STOP ME!!!\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);
    pthread_cancel(thread);
    printf("Requested to cancel the thread\n");
    pthread_join(thread, NULL);
    printf("The thread is stopped.\n");
    return EXIT_SUCCESS;
}
```

На початку імпортуємо потрібні бібліотеки за допомогою директиви #include. Бібліотека stdlib потрібна для використання статусу завершення EXIT_SUCCESS. stdio – бібліотека яка відповідає за введення та виведення даних (stdio – std io, standard input-output). pthread потрібна для роботи з потоками та імпортування потрібних типів даних.

В поток будемо передавати функцію thread_func. Вона повертає значення типу void* – тобто вказівник на об’єкт невизначеного типу. Приймаємо аргумент того ж типу. Змінюємо статус потока за допомогою функції pthread_setcancelstate() на такий, що не дає себе вимкнути. Після цього запускаємо цикл на 5 ітерацій, який виводить одне й те саме повідомлення. Потім знову змінюємо статус, і дозволяємо закінчити поток. Використовуючи функцію pthread_testcancel() залишаємо точку останову – якщо ззовні поток вже намагались зупинити, то на цьому моменті відбувається вихід з функції. Виводимо повідомлення “You will not stop me!!!” та повертаємо NULL.

Точка входу в програму – функція main(), яка приймає аргументами їх кількість – змінну типу int argc – та масив вказівників на char argv. В цій функції створюємо змінну з ідентифікатором потоку – pthread_t thread. Створюємо його за допомогою функції pthread_create – туди передаємо за адресою ідентифікатор, NULL – атрибути потоку за замовчуванням, thread_func – вказівник на функцію, NULL – аргументи до цієї функції. Після цього намагаємося скасувати поток. Виводимо повідомлення, де сповіщуємо про це. Використовуємо pthread_join, для того щоб потік з основною функцією почекав завершення роботи потоку thread. Виводимо повідомлення про те, що thread був зупинений та повертає успішний статус виходу.

Компілюємо:

```
alex@debian:~/Documents/Lab7$ gcc -pthread -Wall -Werror ex1.c -o ex1
```

(gcc – компілятор, –pthread - ключ за допомогою якого підключаємо бібліотеку pthread, –Wall – ввімкнення відображення помилок, –Werror – всі попередження вважаємо помилками, ex1.c – назва файлу який треба скомпілювати, –o – ключ для того, щоб ввести назву виконуваного файлу, ex1 – файл який потрібно запустити).

Запускаємо:

```
alex@debian:~/Documents/Lab7$ gcc -pthread -Wall -Werror ex1.c -o ex1
alex@debian:~/Documents/Lab7$ ./ex1
I'm still running!
I'm still running!
I'm still running!
I'm still running!
YOU WILL NOT STOP ME!!!
Requested to cancel the thread
The thread is stopped.
alex@debian:~/Documents/Lab7$ gcc -pthread -Wall -Werror ex1.c -o ex1
alex@debian:~/Documents/Lab7$ ./ex1
I'm still running!
I'm still running!
I'm still running!
I'm still running!
Requested to cancel the thread
The thread is stopped.
alex@debian:~/Documents/Lab7$ gcc -pthread -Wall -Werror ex1.c -o ex1
alex@debian:~/Documents/Lab7$ ./ex1
Requested to cancel the thread
I'm still running!
I'm still running!
I'm still running!
I'm still running!
The thread is stopped.
alex@debian:~/Documents/Lab7$
```

Бачимо, що кожного разу результат різний, та не завжди відповідає тому, що. здавалося б, повинно бути. Перший випадок – новостворений поток встиг пройти функцію testcancel, до того як в основному потоці була спроба його зупинити. Другий випадок – спроба скасування припала або на момент зміни статусу, або на точку останову. Третій випадок найбільш логічний – новостворений поток пройшов зміну статусу, в цей момент основний намагався його скасувати. Виводиться повідомлення про спробу скасування. Але поток все одно працює. Після цього він зупиняється.

Ці всі проблеми можна вирішити за допомогою механізма м’ютексів, який розглядається в наступних прикладах.

Програмний код другого прикладу:

alex@debian: ~/Documents/Lab7 | alex@debian: ~/Documents/Lab7

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {
    int res;
    int lots_of_threads;
    pthread_t a_thread[NUM_THREADS];
    void *thread_result;
    srand ((unsigned) time (NULL));

    for (lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {
        res = pthread_create(&(a_thread[lots_of_threads]), NULL,
            thread_function, (void *) &lots_of_threads);
        if (res != 0) {
            perror("Thread creation failed");

            exit(EXIT_FAILURE);
        }
    }
```

```c
    printf("Waiting for threads to finish...\n");

    for (lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0; lots_of_threads--) {
        res = pthread_join(a_thread[lots_of_threads], &thread_result);
        if (res == 0)
            printf ("Picked up a thread\n");
        else
            perror ("pthread_join failed");
    }

    printf("All done\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int my_number = *(int *) arg;
    // int rand_num;
    printf("Thread_function is running. Argument was %d\n", my_number);
    // rand_num = 1 + (int) (9.0 * rand () / (RAND_MAX + 1.0));
    printf("Bye from %d\n", my_number);
    pthread_exit(NULL);
}
```

До вище описаних бібліотек додалось дві. time – потрібна для роботи з часом. unistd додає оголошення певних змінних, які потрібні для інших POSIX-залежних бібліотек.

Директивою #define замінюємо всі згадування в тексті напису NUM_THREADS на 6. Тепер це можна підставляти як змінну будь-куди, і компілятор буде заміняти цей напис на число 6.

void *thread_function(void* arg) – прототип функції, яку будемо передавати в поток. Цей рядок потрібен тому що функція main вище, і там відбувається виклик. Компілятори C не підтримують статичне зв'язування, тому потрібні заголовочні файли та прототипи.

В функції thread_function на початку записуємо у змінну аргумент за допомогою касту (cast) в тип «вказівник на int». Також проводимо операцію розіменування вказівника. Після цього бачимо симуляцію обчислень – так як змінна rand_int ніде не використовується, то компілятор попереджає про це, а ключ компіляції –Werror автоматично перетворює це в помилку. Тому ці рядки закоментовані:

```
alex@debian:~/Documents/Lab7$ gcc -pthread -Wall -Werror ex2.c -o ex2
ex2.c: In function ‘thread_function’:
ex2.c:43:9: error: variable ‘rand_num’ set but not used [-Werror=unused-but-set-variable]
   43 |     int rand_num;
      |         ^~~~~~~~
cc1: all warnings being treated as errors
```

Виводимо аргумент на екран, та також виводимо прощавальне повідомлення. Закінчуємо поток.

В головній функції (main) оголошуємо декілька змінних. res – цілочисленне значення, яке буде зберігати в собі код завершення певних операцій. int lots_of_threads – лічильник. Зустрічається два рази, тому виносимо за цикл – це є оптимізацією. phtread_t a_threads[NUM_THREADS] – статичний масив на 6 елементів з ідентифікаторами потоків. void* thread_result – змінна для отримання коду результату від операції злиття потоків. Використовуючи функцію srand встановлюємо рандомайзер на випадкове значення. Після цього створюємо цикл на 6 ітерацій, створюємо 6 потоків, всім передаємо функцію thread_func. Робимо перевірку на код завершення операції створення. Після цього також циклом об’єднуємо всі потоки, виводимо повідомлення та завершуємо програму.

Результати виконання:

```
alex@debian:~/Documents/Lab7$ gcc -pthread -Wall -Werror ex2.c -o ex2
alex@debian:~/Documents/Lab7$ ./ex2
Thread_function is running. Argument was 4
Bye from 4
Waiting for threads to finish...
Thread_function is running. Argument was 5
Bye from 5
Thread_function is running. Argument was 5
Bye from 5
Thread_function is running. Argument was 5
Bye from 5
Thread_function is running. Argument was 5
Bye from 5
Thread_function is running. Argument was 5
Bye from 5
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
```

Цього разу теж можна побачити неочікувану поведінку програми. Цього разу це сталось через те, що, по-перше, не встановлені м’ютекси – цикл встигає змінити значення до ініціалізації змінної у потоці, а змінна передається за адресою. По-друге, злиття встановлене занадто пізно. Взагалі, для таких задач багатопотоковість не потрібна, це лише збільшує об’єм коду та ускладнює програму.

Програмний код третього прикладу:

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

pthread_mutex_t count_mutex;
long count;

void* increment_count(void *arg) {
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);

    return NULL;
}

long get_count() {
    long c;

    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);

    return c;
}
```

```c
int main() {
    pthread_t thread[5];
    pthread_mutex_init(&count_mutex, NULL);
    for (int i = 0; i < 5; i++) {
        pthread_create(&(thread[i]), NULL, increment_count, NULL);
        printf("%ld\n", get_count());
    }
    printf("\n");
    return 0;
}
```

В головній функції створюємо масив ідентифікаторів потоку на 5 елементів. Ініціалізуємо м’ютекс – передаємо за посиланням змінну, яка була оголошена на початку програми. Знову по циклу створюємо потоки, які записуємо в масив. Передаємо їм функцію increment_count. В цій функції збільшується глобальна змінна count на 1. При цьому, ця операція знаходиться між двух операцій з м’ютексами – блокування та розблокування. Це забезпечує атомарність операції – тобто, поток не може отримати значення коли інший редагує те ж значення в цей момент. Після виконання потоком цієх задачі, на екран виводиться результат іншої функції, яка атомарно повертає значення глобальної змінної. Результат роботи програми:

```
alex@debian:~/Documents/Lab7$ ./ex3
0
1
2
3
4
```

Програмний код прикладу 3А:

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

#define LIMIT 16
pthread_mutex_t count_mutex;
long count = 0;

void *increment_count(void *arg) {
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    printf("%ld\n", count);
    pthread_mutex_unlock(&count_mutex);

    return NULL;
}
```

```c
int main(void) {
    pthread_t a_thread[LIMIT];
    pthread_mutex_init(&count_mutex, NULL);

    for (register int i = 0; i < LIMIT; i++) {
        pthread_create(&(a_thread[i]),NULL, increment_count, NULL);
        //printf("%ld\n", get_count());
    }

    for(register int i = 0; i < LIMIT; i++) {
        pthread_join(a_thread[i], NULL);
    }

    pthread_mutex_destroy(&count_mutex);

    return 0;
}
```

Знову створюємо статичний масив з ідентифікаторами потоків, цього разу на 16 елементів. Відразу ініціалізуємо м’ютекс, передаємо за посиланням глобальну змінну count_mutex. Проходимось циклом, створюємо потоки, передаємо їм вказівник на функцію increment_count. Ця функція написана більш правильно, ніж попередня – всі маніпуляції зі змінної зібрані в одному місці, і через це забезпечується стовідсоткова атомарність. Минула програма могла працювати неправильно, але тут все виправляється. Також, м’ютекс знищується за допомогою функції pthread_mutex_destroy, тобто пам’ять очищається, не може виникнути помилка Segmentation Fault, і зміна і є регістерною – тобто може передаватись відразу копією.

Програмний код прикладу 4:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>

#define SIZE_I 2
#define SIZE_J 2

float X[SIZE_I][SIZE_J];
float S[SIZE_I][SIZE_J];

int count = 0; // глобальний лічильник

struct DATA_ {
    double x;
    int i;
    int z;
};

typedef struct DATA_ DATA;

// Оголошення змінної м'ютексу
pthread_mutex_t lock;

// Функція для обчислень
double f(double x) { return x*x; }
```

```c
// Потокова функція для обчислень
void *calc_thr(void *arg) {
    DATA* a = (DATA*) arg;
    X[a->i][a->z] = f(a->x);

    // установка блокування
    pthread_mutex_lock(&lock);

    // зміна глобальної змінної
    count++;

    // зняття блокування
    pthread_mutex_unlock(&lock);
    free(a);

    return NULL;
}

// Потокова функція для введення
void *input_thr(void *arg) {
    DATA* a = (DATA*) arg;

    pthread_mutex_lock(&lock);

    printf("S[%d][%d]:", a->i, a->z);
    scanf("%f", &S[a->i][a->z]);

    pthread_mutex_unlock(&lock);

    free(a);
```

```
    return NULL;
}

int main() {
    //масив ідентифікаторів потоків
    pthread_t thr[SIZE_I * SIZE_J];

    // ініціалізація м'ютекса
    pthread_mutex_init(&lock, NULL);

    DATA *arg;

    // Введення даних для обробки
    for (int i = 0; i < SIZE_I; i++) {
        for (int z = 0; z < SIZE_J; z++) {
            arg = malloc(sizeof(DATA));
            arg->i = i;
            arg->z = z;

            // створення потоку для введення елементів матриці
            pthread_create(&thr[i * SIZE_J + z], NULL, input_thr, (void*) arg);
        }
    }

    // Очікування завершення усіх потоків введення даних
    // ідентифікатори потоків зберігаються у масиві thr

    for (int i = 0; i < SIZE_I * SIZE_J; i++) {
        pthread_join (thr[i], NULL);
    }
```

```
    // Обчислення елементів матриці
    pthread_t thread;
    printf("Start calculation\n");
    for (int i = 0; i < SIZE_I; i++) {
        for (int z = 0; z < SIZE_J; z++) {
            arg = malloc(sizeof(DATA));
            arg->i = i;
            arg->z = z;
            arg->x = S[i][z];

            // створення потоку для обчислень
            pthread_create (&thread, NULL, calc_thr, (void *) arg);

            // переведення потоку у режим від'єнання
            pthread_detach(thread);
        }
    }

    do {
        // Основний процес "засинає" на 1с
        // Перевірка стану обчислень
        sleep(1);
        printf("finished %d threads.\n", count);
    } while (count < SIZE_I * SIZE_J);
```

```
    // Виведення результатів
    for (int i = 0; i < SIZE_I; i++) {
        for (int z = 0; z < SIZE_J; z++) {
            printf("X[%d][%d] = %f\t", i, z, X[i][z]);
        }
        printf("\n");
    }

    // видалення м'ютекса
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Досить елементарна програма, майже всі моменти були пояснені в минулих прикладах. Суть цього прикладу – обрахунок матриці 2x2 за заданими значеннями. Кожне число множиться саме на себе. Але декілька моментів треба пояснити. struct – це оголошення структури. В структурі DATA_ наявні цілочисельні поля, які означають індекси, і також є поле яке зберігає значення дробового типу. Ключове слово typedef struct означає, що структуру можна використовувати, не уточнючи кожного разу перед цим що це структура. Звернення до полів відбувається через оператор "->". Команда free

звільнює динамічно виділену пам’ять за заданою адресою. Команда malloc виділяє вказану кількість байтів динамічної пам’яті. Detach потоків не чекає злиття – процес переривається щойно один дійшов до цієї команди. В програмі наявні атомарні функції для зчитування з клавіатури даних та обрахунку значень. Єдиний недолік – зчитування даних повинно бути однопоточне. Інакше порядок зчитуваних значень буде переплутаний. Результат програми:

```
alex@debian:~/Documents/Lab7$ gcc -pthread -Wall -Werror ex4.c -o ex4
alex@debian:~/Documents/Lab7$ ./ex4
S[0][0]: 5
S[1][0]: 3
S[0][1]: 6
S[1][1]: 1
Start calculation
Finished 4 threads.
X[0][0] = 25.000000     X[0][1] = 36.000000
X[1][0] = 9.000000      X[1][1] = 1.000000
alex@debian:~/Documents/Lab7$
```

Програмний код прикладу 5:

```c
#include <error.h>
#include <inttypes.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdint.h>
#include <stdio.h>

void *WriteToFile(void *);
sem_t psem;
FILE *in;

int main(int argc, char* argv[]) {
    pthread_t tidA, tidB;

    in = fopen("result.txt", "w");

    int number = 100;

    sem_init(&psem, 0, 0);
    sem_post(&psem);

    pthread_create(&tidA, NULL, &WriteToFile, (void *) &number);
    pthread_create(&tidB, NULL, &WriteToFile, (void *) &number);

    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);

    fclose(in);

    sem_destroy(&psem);
```

```c
    sem_destroy(&psem);

    return 0;
}

void *WriteToFile(void *f) {
    int max = *((int *) f);
    for (int i = 0; i <= max; i++) {
        sem_wait(&psem);
        fprintf(in, "-writetofilecounter i=%d \n", i);
        sem_post(&psem);
    }
    return NULL;
}
```

На початку оголошується прототип функції запису в файл. Потім, оголошується змінна семафору. Також, оголошується змінна файлу. В головній функції оголошується 2 потоки. Після цього відкривається файл в режимі запису. Ми хочемо записати числа від 1 до 100, тому ініціалізуємо змінну з цим числом. Ініціалізовуємо семафор зі значенням 0. Додаємо йому 1, щоб хоча б один процес міг пройти. Після цього запускаємо два потоки в потрібну нам функцію. Там забираємо аргумент, та циклом проходимось і записуємо числа в файл. Тут нічого складного немає, тому що в даному випадку семафор працює як м'ютекс – значення семафору не перевищує 1. Після запису у файл об'єднуємо потоки, закриваємо файл та знищуємо семафор.

Результат виконання програми:

```
alex@debian:~/Documents/Lab7$ gcc -pthread -Wall -Werror ex5.c -o ex5
alex@debian:~/Documents/Lab7$ ./ex5
alex@debian:~/Documents/Lab7$ cat result.txt
-writetofilecounter i=0
-writetofilecounter i=1
-writetofilecounter i=2
-writetofilecounter i=3
-writetofilecounter i=4
-writetofilecounter i=5
-writetofilecounter i=6
-writetofilecounter i=7
-writetofilecounter i=8
-writetofilecounter i=9
-writetofilecounter i=10
-writetofilecounter i=11
-writetofilecounter i=12
-writetofilecounter i=13
-writetofilecounter i=14
-writetofilecounter i=15
-writetofilecounter i=16
-writetofilecounter i=17
-writetofilecounter i=18
-writetofilecounter i=19
-writetofilecounter i=20
-writetofilecounter i=21
-writetofilecounter i=22
```

Наступним завданням є запис у файл використовуючи стандартне введення з клавіатури. Для цього використаємо оператор >, який означає перенаправлення виводу, та утиліту cat, виведемо результат на екран:

```
alex@debian:~/Documents/Lab7$ cat > myfile.txt
Group: TP-12
Name: Alex
Surname: Kovalyov
Patronymic: Oleksiyovuch
alex@debian:~/Documents/Lab7$ cat myfile.txt
Group: TP-12
Name: Alex
Surname: Kovalyov
Patronymic: Oleksiyovuch
alex@debian:~/Documents/Lab7$
```

Додамо в файл інформацію про хобі:

```
alex@debian:~/Documents/Lab7$ echo -e "Hobby: Programming\n" >> myfile.txt
alex@debian:~/Documents/Lab7$ cat myfile.txt
Group: TP-12
Name: Alex
Surname: Kovalyov
Patronymic: Oleksiyovuch
Hobby: Programming

alex@debian:~/Documents/Lab7$
```

Можна перевірити роботу конвеєра прикладом з роботи – знайти, скільки є рядків, в яких наявне слово “Group”. Для цього використаємо утиліти grep для пошуку та wc для підрахунку.

```
alex@debian:~/Documents/Lab7$ cat myfile.txt | grep "Group" | wc -l
1
alex@debian:~/Documents/Lab7$ cat myfile.txt
Group: TP-12
Name: Alex
Surname: Kovalyov
Patronymic: Oleksiyovuch
Hobby: Programming
```

Результат правильний, є тільки один такий рядок.

Так можна передавати результати однієї команди іншій безкінечно. Це дуже корисно для системних адміністраторів, комбінуючи команди можна діставати абсолютно будь-яку інформацію.

Завдання – створити канал. Для цього можем використати такий тип даних, як FIFO. Це канал, який працює за принципом черги, тобто “First In, First Out”. [2]

Створюємо чергу за допомогою команди mkfifo та перенаправляємо туди текст:

```
alex@debian:~/Documents/Lab7$ mkfifo pipe
alex@debian:~/Documents/Lab7$ cat > pipe
some text
some text again
```

Після створення процес не можна перервати, тому що інформація повинна вийти. Перейдемо на другий термінал та спробуємо прочитати інформацію звідти:

```
alex@debian:~/Documents/Lab7$ cat pipe
some text
some text again
alex@debian:~/Documents/Lab7$
```

```
alex@debian:~/Documents/Lab7$ cat > pipe
some text
some text again
alex@debian:~/Documents/Lab7$
```

Як бачимо, в другому терміналі з’явилась інформація, яка була введена, а перший термінал звільнився. Канал працює, експеримент вдалий.

Контрольні запитання:

1) **У чому полягає відмінність потоку та процесу?**
   Реалізація потоків та процесів відрізняються в різних операційних системах, але загалом потік міститься всередині процесу і різні потоки одного процесу спільно розподіляють деякі ресурси, у той час як різні процеси ресурси не розподіляють.
2) **Які вам відомі типи потоків?**
   Користувацькі потоки та потоки ядра.
3) **Які вам відомі основні функції при роботі з потоками?**
   pthread_create, phtread_join, phtread_detach, phtread_self, phtread_exit, phtread_cancel, phtread_setcancelstate, phtread_testcancel та інші.
4) **Які вас відомі засоби синхронізації потоків?**
   М’ютекси, семафори.
5) **Які стандартні потоки введення/виведення?**
   stdin – дескриптор 0 – стандартне введення з клавіатури. stdout – дескриптор 1 – стандартне виведення, екран. stderr – дескриптор 2 – стандартне виведення помилок (екран).
6) **Які види перенаправлення вам відомі?**
   Звичайне, звичайне з доповненням, та перенаправлення конвеєром.
7) **Що таке канали?**
   Канали – це механізм міжпроцесної взаємодії, який перенаправляє результат однієї програми в іншу. Найвідомішим представником каналів є конвеєр – працює з оператором «|».

Висновок: за результатами виконання цієї лабораторної роботи було ознайомлено з такими поняттями як поток, були набуті навички в управлінні ними. Була освоєна концепція каналів, перенаправлення потоків, потоків виведення та введення. Була проведена робота з системними викликами потоків.

Додаткові джерела:

1) Немет, Эви, Снайдер – Unix и Linux: руководство системною администратора, 5-е изд.: Пер. с англ. - СПб. : ООО "Диалектика", 2020. - 1 168 с. : ил. - Парал. тит. англ.
2) PortaOne Education Centre – Files lection