

## Bài thí nghiệm 2

# KỸ THUẬT LẬP TRÌNH NÂNG CAO CHO PHẦN MỀM NHÚNG: TIỀN TRÌNH, LUỒNG, ĐỒNG BỘ ĐA LUỒNG

### Mục đích

- Nắm được các kỹ thuật lập trình nâng cao trong phát triển phần mềm nhúng, bao gồm:
  - Các kỹ thuật tạo lập/gọi tiến trình mới, cơ chế giao tiếp giữa các tiến trình
  - Các kỹ thuật lập trình với luồng, đa luồng
  - Kỹ thuật đồng bộ đa luồng sử dụng mutex và semaphore
- Các bài thí nghiệm được thực hiện bằng lập trình C trên nền tảng Arm Linux (sử dụng thiết bị Raspberry Pi 4), cũng như vận dụng bằng các ngôn ngữ hướng đối tượng khác như Java.

### Chuẩn bị

- Máy tính PC dùng Windows hoặc Ubuntu có các công cụ cần thiết: Visual code, ssh extension (xem bài mở đầu)
- KIT Raspberry Pi 4

### Tiến hành thí nghiệm

#### 1. Các kỹ thuật lập trình với tiến trình (process)

##### Bài 1.1. Số hiệu tiến trình

**Bước 1.** Viết một tiến trình đơn giản với mã nguồn như sau (File **process1.c**):

```
#include<stdio.h>// standard input / output functions
#include <unistd.h>
#include <time.h>
int main(int argc, char** argv)
{
    printf("\nMa tien trinh dang chay : %d", (int)getpid());
    printf("\nMa tien trinh cha : %d", (int)getppid());
    while (1)
    {
        printf("\nRunning...");
        usleep(500000);
    }
}
```

Trong tiến trình này, sử dụng các hàm getpid(), getppid() để lấy định danh của tiến trình đang chạy (chứa lời gọi hàm) và định danh của tiến trình cha của nó. Các định danh tiến trình là tham số quan trọng để quản lý và giúp các tiến trình giao tiếp với nhau qua cơ chế signal.

**Bước 2.** Biên dịch, thực hiện chương trình trên máy tính hoặc KIT, quan sát kết quả.

gcc -o process1 process1.c

**Bước 3.** Dùng lệnh kill gửi signal tới tiến trình. Ví dụ kết thúc tiến trình:

kill SIGTERM <PID>

kill -9 <PID>

The screenshot shows a Visual Studio Code window with a C program named `process1.c` and a terminal window. The C program is as follows:

```

1 #include<stdio.h> // standard input / output functions
2 #include<unistd.h>
3 #include<time.h>
4 int main(int argc, char** argv)
5 {
6     printf("\nMa tien trinh dang chay : %d", (int)getpid());
7     printf("\nMa tien trinh cha : %d", (int)getppid());
8     while (1)
9     {
10         printf("\nRunning-");
11         usleep(500000);
12     }
13 }
14

```

The terminal window shows the execution of the program on a Raspberry Pi. The output is as follows:

```

pi@raspberrypi:~/code/bth3 $ ./process1
Ma tien trinh dang chay : 7304
Ma tien trinh cha : 6813
Running_
Running_
Running_
Running_
Running_
Running_
Running_
Running_
Running_
Killed
pi@raspberrypi:~/code/bth3 $

```

The terminal also shows the command `kill -9 7304` being executed, which kills the process.

## Bài 1.2. Tạo lập tiến trình mới

Khi một chương trình muốn gọi một chương trình khác, cơ chế tạo lập tiến trình mới thường được áp dụng. Có thể thực hiện điều này với lập trình C (trên hệ điều hành Linux) với các cách tạo lập trình mới gồm:

a) Gọi tiến trình mới bằng hàm `system()`

Viết chương trình với mã nguồn như sau (File `process2a.c`)

```

#include<stdio.h> // standard input / output functions
#include<unistd.h>
#include<time.h>
int main(int argc, char** argv)
{
    printf("Ma tien trinh goi ham system(): %d\n", (int)getpid());
    printf("Chay tien trinh moi voi ham system()\n");
    system("./process1");
    return(0);
}

```

Biên dịch, chạy chương trình và quan sát kết quả thực hiện

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <time.h>
5
6  int main (int argc, char** argv)
7  {
8      printf("Ma tien trinh goi ham system(): %d\n", (int) getpid());
9      printf("Chay tien trinh moi voi ham system()\n");
10     system("./process1");
11     return(0);
12 }

```

```

pi@raspberrypi:~/code/bth3 $ ./process1_3_2
Ma tien trinh goi ham system(): 8264
Chay tien trinh moi voi ham system()

Ma tien trinh dang chay : 8266
Ma tien trinh cha : 8265
Running..
Running..
Running..
Running..
Running..
Running..
Running..
Running..
Running..
Running..
^Cpi@raspberrypi:~/code/bth3 $

```

b) Thay thế tiến trình mới bằng hàm `exec()`

Viết chương trình với mã nguồn sau (File `process2b.c`)

```

#include<stdio.h> // standard input / output functions
#include <unistd.h>
#include <time.h>
int main(int argc, char** argv)
{
    printf("Ma tien trinh exec() dang chay : %d\n", (int) getpid());
    char *args[] = {"./process1", NULL};
    printf("Chay tien trinh moi voi ham exec()\n");
    execvp(args[0], args);
    return(0);
}

```

Biên dịch, chạy chương trình và quan sát kết quả thực hiện

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <time.h>
5
6 int main (int argc, char** argv)
7 {
8     printf("Ma tien trinh exec() dang chay : %d\n", (int) getpid());
9     char *args[] = {"/.process1", NULL};
10    printf("Chay tien trinh moi voi ham exec()\n");
11    execvp(args[0], args);
12    return(0);
13 }
    
```

```

pi@raspberrypi:~/code/bth3 $ ./process1_3
Ma tien trinh exec() dang chay : 8633
Chay tien trinh moi voi ham exec()

Ma tien trinh dang chay : 8633
Ma tien trinh cha : 6813
Running_
Running_
Running_
Running_
^C
pi@raspberrypi:~/code/bth3 $
    
```

- c) Gọi tiến trình mới sử dụng lập trình JAVA. Viết chương trình java với mã nguồn sau: (File: process2c.java)

```

public class Program {
    public static void main(String argv[])
    {
        String args[] = {"ls -al"};
        try {
            Process p = Runtime.getRuntime().exec(args);
            //Lấy output tiến trình được gọi
            BufferedReader in = new BufferedReader(new
InputStreamReader(p.getInputStream()));
            String line = in.readLine();
            while (line != null) {
                System.out.println("Found: " + line);
                line = in.readLine();
            }
        } catch (Exception e) {
            System.err.println("Some error occurred : " + e.toString());
        }
    }
}
    
```

Biên dịch, chạy chương trình và quan sát kết quả.

### Bài 1.3. Giao tiếp giữa các tiến trình

Sử dụng cơ chế đón bắt và xử lý signal (tín hiệu) để giao tiếp giữa các tiến trình (truyền tín hiệu/sự kiện từ 1 tiến trình này đến 1 tiến trình khác)

Viết chương trình như minh họa sau đây (file **process2.c**)

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int count = 0;
//Hàm xử lý được gọi khi có tín hiệu SIGUSR1 gửi tới tiến trình
void handler(int signal_number)
{
    FILE *fid = fopen("output.txt", "a"); //Ghi thông tin ra tệp. Nếu là
    ~/output.txt thì sẽ gây ra lỗi Segmentation fault
    fprintf(fid, "\nNhan duoc tin hieu SIGUSR1 lan thu %d", count++);
    fclose(fid);
}
int main()
{
    struct sigaction sa; //Khai báo một biến cấu trúc sigaction
    printf("The process ID is %d\n", (int)getpid());
    printf("The parent process ID is %d\n", (int)getppid());
    //Thiết lập signal handler
    memset(&sa, 0, sizeof(sa));
    //Gán con trỏ hàm xử lý signal cho trường sa_handler của biến cấu trúc sa
    sa.sa_handler = &handler;
    //Đăng ký cấu trúc sa cho xử lý tín hiệu (signal) SIGUSR1
    sigaction(SIGUSR1, &sa, NULL);
    //Chương trình chính liên tục in ra chữ A
    while (1)
    {
        printf("A");
        sleep(1);
    }
    return 0;
}
```

- Biên dịch và chạy chương trình trên Raspberry Pi:

gcc -o process2 process2.c

- Thực thi tiến trình này, quan sát kết quả, xem số hiệu pid.
- Mở cửa sổ terminal khác, gửi tín hiệu SIGUSR1 đến tiến trình này bằng lệnh linux:

**kill -SIGUSR1 pid hoặc kill -10 pid**

(pid là số hiệu tiến trình process2 đang chạy. Nếu ko có dấu – thì hệ thống sẽ hủy tiến trình theo mặc định)

- Thử gửi tín hiệu này vài lần và mở file output.txt (Thư mục \$HOME) quan sát kết quả.
- Kết quả thực hiện

The screenshot shows a Visual Studio Code editor with two main panes. The left pane displays a C program named `process2.c` with the following code:

```

1 #include <signal.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <unistd.h>
6 #include <stdlib.h>
7 int count = 0;
8 //Hàm xử lý được gọi khi có tín hiệu SIGUSR1 gửi tới tiến trình
9 void handler(int signal_number)
10 {
11     FILE *fid = fopen("output.txt", "a"); //Ghi thông tin ra file
12     fprintf(fid, "\nNhan duoc tin hieu SIGUSR1 lan thu %d", count);
13     fclose(fid);
14 }
15 int main()

```

The right pane shows the output of the program, which is a file named `output.txt` containing the following text:

```

1
2 Nhan duoc tin hieu SIGUSR1 lan thu 0
3 Nhan duoc tin hieu SIGUSR1 lan thu 1
4 Nhan duoc tin hieu SIGUSR1 lan thu 2
5 Nhan duoc tin hieu SIGUSR1 lan thu 3
6 Nhan duoc tin hieu SIGUSR1 lan thu 4

```

At the bottom, the terminal shows the execution of the program and the use of the `kill` command to send signals to the process.

```

pi@raspberrypi:~/code/bth3 $ ./process2
The process ID is 17651
The parent process ID is 8958

pi@raspberrypi:~/code/bth3 $ kill -SIGUSR1 17651
pi@raspberrypi:~/code/bth3 $ kill -SIGUSR1 17651
pi@raspberrypi:~/code/bth3 $ kill -SIGUSR1 17651
pi@raspberrypi:~/code/bth3 $ kill -SIGUSR1 17651
pi@raspberrypi:~/code/bth3 $ kill -SIGUSR1 17651
pi@raspberrypi:~/code/bth3 $

```

- **Ghi chú:** Có thể sử dụng hàm `kill()` trong lập trình C để gửi tín hiệu đến một tiến trình. Thư viện: `#include <signal.h>`

```
int kill(pid_t pid, int sig);
```

**Bài tập bổ sung :** Viết một chương trình c sử dụng hàm này, tham số pid của tiến trình muốn gửi signal đến được truyền từ dòng lệnh. Thực hiện chương trình này để gửi tín hiệu SIGUSR1 đến tiến trình process2 (file sendprocess2.c)

```

#include <signal.h>
#include <inttypes.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <errno.h>

int main(int argc, char** argv)
{
    int sig;
    pid_t pid;
    intmax_t xmax;
    char *tmp;
    errno=0;
    if(argc==3)
    {
        sig = atoi(argv[1]);
        xmax = strtoumax(argv[2], &tmp, 10);
        if(errno != 0 || tmp == argv[2] || *tmp != '\0' || xmax != (pid_t)xmax)
        {
            fprintf(stderr, "Bad PID!\n");
        }
        else
        {
            pid = (pid_t)xmax;
        }
        printf("\nkill %d %d\n", (int)pid, sig);
        kill(pid, sig);
    }
}

```

```
    }  
    return 0;  
}
```

## 2. Các kỹ thuật lập trình với luồng (thread)

Các kỹ thuật lập trình với luồng được minh họa bằng C sử dụng thư viện pthread trên Linux.

## Bài 2.1. Thực hiện tạo luồng đơn giản

Viết chương trình với mã nguồn sau thực hiện tạo luồng với hàm pthread\_create

```
(File: thread1.c)

#include <stdio.h>
#include <pthread.h>
//Hàm xử lý của luồng thực hiện liên tục in chữ x
void* printx(void* unused)
{
    while (1)
    {
        fputc('x', stdout);
    }
    return NULL;
}

int main(int argc, char** argv)
{
    pthread_t thread_id;
    //Tạo ra một luồng mới với hàm xử lý luồng là printx
    pthread_create(&thread_id, NULL, &printx, NULL);
    //Chương trình chính liên tục in chữ o
    while (1)
    {
        fputc('o', stdout);
    }
    return 0;
}

//Trong ví dụ này cả chương trình chính là luồng đều chạy lặp vô hạn
```

- Biên dịch và chạy chương trình trên KIT và quan sát kết quả

```
pi@raspberrypi:~/code/bth3
pi@raspberrypi:~/code/bth3 $ gcc thread1.c -o thread1 -lpthread
pi@raspberrypi:~/code/bth3 $ ./thread1
```

## Bài 2.2. Truyền tham số cho luồng

- Chú ý: Hàm xử lý của luồng có kiểu tham số là void\*, vì vậy để truyền nhiều thành phần dữ liệu cho luồng cần tạo một cấu trúc (struct), và truyền cho hàm xử lý của luồng tham số là biến cấu trúc này.

- Ví dụ sau minh họa tạo ra 2 luồng cùng sử dụng hàm xử lý in một số lượng ký tự ra màn hình, với tham số truyền vào là ký tự và số lượng muốn in.

File: **thread2.c**

```
#include <pthread.h>
#include <stdio.h>
//Cau truc la tham so cho ham xu ly luong (ham char_print)
struct char_print_parms
{
    char character; //Ky tu muon in
    int count; //So lan in
};
//Ham xu ly cua thread
//In ky tu ra man hinh, duoc cho boi tham so la mot con tro den cau truc du lieu
tren
void* char_print(void* params)
{
    //Tham so truyen vao la kieu void* duoc ep thanh kieu nhu struct da khai bao
    struct char_print_parms* p = (struct char_print_parms*) params;
    int i;
    int n = p->count; //Bien chua so lan in ra
    char c = p->character; //Bien chua ma ky tu muon in ra
    for (i = 0; i < n; i++)
        fputc(c, stdout); //Ham in 1 ky tu ra thiet ra chuan
    return NULL;
}
int main(int argc, char** argv)
{
    pthread_t thread1_id, thread2_id; //Khai bao 2 bien dinh danh luong
    struct char_print_parms p1, p2; //2 bien tham so truyen cho ham xu ly cua
thread
    //Tao 1 thread in 30000 chu 'x'
    p1.character = 'x';
    p1.count = 30000;
    pthread_create(&thread1_id, NULL, &char_print, &p1);
    //Tao 1 thread khac in ra 20000 chu 'o'
    p2.character = 'o';
    p2.count = 20000;
    pthread_create(&thread2_id, NULL, &char_print, &p2);
    //Dam bao thread1 da ket thuc
    pthread_join(thread1_id, NULL);
    //Dam bao thread2 da ket thuc
    pthread_join(thread2_id, NULL);
    // Now we can safely return.
    return 0;
}
```

- Biên dịch, chạy chương trình và quan sát kết quả

### Bài 2.3. Sử dụng đa luồng cho chương trình giao tiếp button, led



- Sử dụng breadboard và linh kiện cần thiết (LED, điện trở, dây) để lắp 1 mạch ghép nối GPIO gồm cụm 4 LED và 2 nút bấm K1, K2 cho Raspberry Pi 4.
- Sử dụng lập trình đa luồng, viết một chương trình trên Raspberry Pi thực hiện hiệu ứng led đuổi, sử dụng nút bấm K1, K2 để thay đổi (giảm/tăng) tốc độ của hiệu ứng led đuổi.

Mô tả:

- Chương trình chính (main thread) thực hiện hiệu ứng led đuổi trong một vòng lặp vô hạn, hiệu ứng dựa trên viết bật/tắt từng led với thời gian trễ (delay) thích hợp (giả sử là t milisecond, ban đầu mặc định là 1000ms=1s)
  - Thời gian trễ nói trên có thể điều chỉnh (tăng/giảm thích hợp) khi bấm nút K1, K2.
  - Giao tiếp nút bấm (K1, K2) sử dụng theo cơ chế polling (thăm dò), cần sử dụng một luồng riêng để thực hiện công việc này (song song với công việc chính là điều khiển hiệu ứng nháy led), hàm xử lý của luồng sẽ đọc K1, K2 có được ấn để thay đổi giá trị t tương ứng.
- Tham khảo mã nguồn sau, biên dịch và chạy chương trình trên KIT Raspberry Pi 4  
(File: **threadbuttonled.c**)

```

/*****
* Chương trình:
* main: Thực hiện điều khiển led chạy đuổi
* thread: Thực hiện đọc trạng thái (polling) nút bấm để thay đổi tốc độ led
*****/
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/time.h>
#define MAX 64

//Định nghĩa các giá trị chân board
#define K1 2
#define K2 3
#define LED1 4
#define LED2 5
#define LED3 6
#define LED4 7
#define ON 1
#define OFF 0

static int t = 500;
int GPIO_in[2] = {K1, K2};
int GPIO_out[4] = {LED1, LED2, LED3, LED4};
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

```

```
//Hàm delay
void sleepms(int ms);

//Hàm khởi tạo các chân input cho nút bấm và output cho đèn
int initGPIO(int pin_in[], int pin_out[]);

//Hàm xử lý trạng thái đèn
int digitalWrite(int GPIO_pin, int value);

//Hàm polling đọc giá trị nút bấm
void* btn_polling(void* param);

int main() {
    initGPIO(GPIO_in, GPIO_out);
    int K1_pin = K1;
    int K2_pin = K2;

    //Tạo 2 luồng con cho mỗi nút bấm
    pthread_t K1_thread_id;
    pthread_t K2_thread_id;
    K1_thread_id = pthread_create(&K1_thread_id, NULL, btn_polling, &K1_pin);
    K2_thread_id = pthread_create(&K2_thread_id, NULL, btn_polling, &K2_pin);
    //Luồng master chạy riêng điều khiển chạy Led
    int led_no = 0;
    while(1) {
        digitalWrite(GPIO_out[led_no], ON);
        sleepms(t);
        digitalWrite(GPIO_out[led_no], OFF);
        led_no++;
        if(led_no == 4) {
            led_no = 0;
        }
    }
    pthread_join(K1_thread_id, NULL);
    pthread_join(K2_thread_id, NULL);
    return 0;
}

void sleepms(int ms) {
    usleep(1000 * ms);
    return;
}

//Hàm khởi tạo các chân input cho nút bấm và output cho đèn
int initGPIO(int pin_in[], int pin_out[]) {
    int n_pin_in = 2;
    int n_pin_out = 4;
    //Tạo con trỏ file để xử lý các sysfs tương ứng với các chân
    FILE *fp = NULL;
    char setValue[4], GPIO_direction[MAX];
```

```

        //Đặt các chân làm input dành cho nút bấm thông qua file direction
        for(int i = 0; i < n_pin_in;i++) {
            sprintf(GPIO_direction, "/sys/class/gpio/gpio%d/direction",
pin_in[i]);
            if((fp = fopen(GPIO_direction, "wb")) == NULL) {
                printf("Cannot open direction of gpio %d", pin_in[i]);
                return 1;
            }
            strcpy(setValue, "in");
            fwrite(&setValue, sizeof(char), 3, fp);
            fclose(fp);
        }

        //Đặt các chân làm output dành cho các Led thông qua file direction
        for(int i = 0; i < n_pin_out;i++) {
            sprintf(GPIO_direction, "/sys/class/gpio/gpio%d/direction",
pin_out[i]);
            if((fp = fopen(GPIO_direction, "wb")) == NULL) {
                printf("Cannot open direction of gpio %d", pin_out[i]);
                return 1;
            }
            strcpy(setValue, "out");
            fwrite(&setValue, sizeof(char), 3, fp);
            fclose(fp);
        }
        return 0;
    }

//Hàm xử lý trạng thái đèn
int digitalWrite(int GPIO_pin, int value) {
    FILE *fp = NULL;
    char setValue[4], GPIO_value[MAX];
    sprintf(GPIO_value, "/sys/class/gpio/gpio%d/value", GPIO_pin);
    //Mở file value của các chân Led và truyền vào giá trị mức cao hoặc thấp
    if((fp = fopen(GPIO_value, "wb")) == NULL) {
        printf("Cannot open value of gpio %d", GPIO_pin);
        return 1;
    }
    sprintf(setValue, "%d", value);
    fwrite(&setValue, sizeof(char), 1, fp);
    fclose(fp);
    return 0;
}

//Hàm polling đọc giá trị nút bấm
void* btn_polling(void* param) {
    int* K_pin = (int*) param;
    printf("GPIO of K1 = %d\n", *K_pin);

    FILE *K_file = NULL;

```

```

char setValue[4], K_value[MAX];
sprintf(K_value, "/sys/class/gpio/gpio%d/value", *K_pin);

int K_old = 0;
//Dùng polling để đọc giá trị input từ file value của chân tương
ứng
while(1) {
    if((K_file = fopen(K_value, "rb")) == NULL) {
        printf("Cannot open value of gpio %d\n", *K_pin);
        exit(1);
    }
    //Đọc giá trị và chuyển về trạng thái để so sánh với các trạng
    thái trước

    fread(&setValue, sizeof(char), 1, K_file);
    int K_cur = (int)setValue[0] - 48;
    if(K_old != K_cur) {
        K_old = K_cur;

        //Dùng khóa mutex để khóa giá trị t cho riêng luồng này
        pthread_mutex_lock(&lock);
        if(*K_pin == K1) {
            t = t + 100;
            printf("Increase t = %d\n", t);
        } else {
            t = t - 100;
            if(t < 100) {
                t = 100;
            }
            printf("Decrease t = %d\n", t);
        }
        sleepms(100);
        pthread_mutex_unlock(&lock);
    }
    fclose(K_file);
}
return NULL;
}

```

### 3. Đồng bộ luồng

- 2 kỹ thuật đồng bộ luồng cơ bản: mutex và semaphore. Bài thí nghiệm này sẽ thực hiện minh họa kỹ thuật đồng bộ sử dụng khóa mutex với lập trình C trên Linux.

#### Bài 3.1. Đồng bộ luồng sử dụng khóa Mutex

- Viết chương trình sử dụng đồng bộ luồng bằng khóa Mutex với mã nguồn sau

File: **mutex.c**

```

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

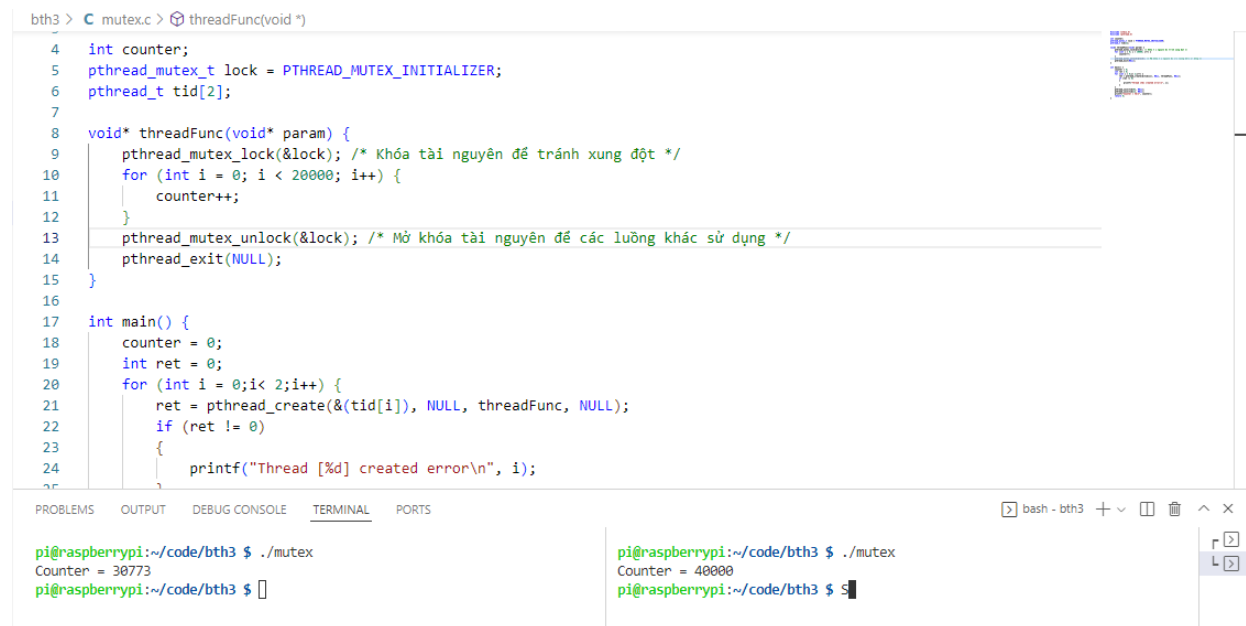
```

```
int counter;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_t tid[2];

void* threadFunc(void* param) {
    pthread_mutex_lock(&lock); /* Khóa tài nguyên để tránh xung đột */
    for (int i = 0; i < 20000; i++) {
        counter++;
    }
    pthread_mutex_unlock(&lock); /* Mở khóa tài nguyên để các luồng khác sử dụng */
    pthread_exit(NULL);
}

int main() {
    counter = 0;
    int ret = 0;
    for (int i = 0; i < 2; i++) {
        ret = pthread_create(&tid[i], NULL, threadFunc, NULL);
        if (ret != 0)
        {
            printf("Thread [%d] created error\n", i);
        }
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    printf("Counter = %d\n", counter);
    return 0;
}
```

- Trong chương trình trên, nếu như không sử dụng đồng bộ luồng bằng khóa mutex để ngăn xung đột tài nguyên, giá trị của biến counter sau khi thực thi xong hai luồng sẽ có kết quả không như mong muốn (mỗi luồng thực hiện 20000 lần đếm tăng biến counter thêm 1 đơn vị), vì có trường hợp cả hai luồng cùng ghi đè giá trị lên biến tại cùng một thời điểm. Tuy nhiên, với sự có mặt của khóa mutex, biến counter sẽ lần lượt chỉ được truy cập bởi một luồng duy nhất tại một thời điểm, điều đó đảm bảo cả luồng sẽ thực hiện 40000 lần tăng giá trị biến counter (giá trị biến counter cuối cùng là 40000).
- Kết quả như hình dưới phản ánh đúng điều vừa trình bày ở trên với hình bên trái là kết quả của biến counter khi không có sự bảo vệ của khóa mutex và hình bên phải là kết quả đúng nhờ áp dụng khóa mutex



```

bth3 > C mutex.c > threadFunc(void *)
4  int counter;
5  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
6  pthread_t tid[2];
7
8  void* threadFunc(void* param) {
9      pthread_mutex_lock(&lock); /* Khóa tài nguyên để tránh xung đột */
10     for (int i = 0; i < 20000; i++) {
11         counter++;
12     }
13     pthread_mutex_unlock(&lock); /* Mở khóa tài nguyên để các luồng khác sử dụng */
14     pthread_exit(NULL);
15 }
16
17 int main() {
18     counter = 0;
19     int ret = 0;
20     for (int i = 0; i < 2; i++) {
21         ret = pthread_create(&tid[i], NULL, threadFunc, NULL);
22         if (ret != 0)
23             printf("Thread [%d] created error\n", i);
24     }
25 }

```

```

pi@raspberrypi:~/code/bth3 $ ./mutex
Counter = 30773
pi@raspberrypi:~/code/bth3 $
pi@raspberrypi:~/code/bth3 $ ./mutex
Counter = 40000
pi@raspberrypi:~/code/bth3 $

```

## Bài 3.2. Đồng bộ luồng bằng mutex với chương trình viết bằng Java

Lập trình đồng bộ luồng bằng mutex với chương trình viết bằng Java.

Sử dụng **ReentrantLock** trong Java

```

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class mutex {
    private final Lock lock = new ReentrantLock(true);
    private int counter = 0;

    /* Chương trình khởi tạo 2 luồng cùng truy cập trực tiếp và thay đổi giá trị của đối tượng Counter */
    public static void main(String[] args) {
        try {
            Thread thread1 = new Thread(new Runnable() {
                @Override
                public void run() {
                    lock.lock();
                    try {
                        for (int i = 0; i < 20000; i++) {
                            counter++;
                        }
                    } catch (Exception e) {
                        // handle the exception
                    } finally {
                        lock.unlock();
                    }
                }
            });
            Thread thread2 = new Thread(new Runnable() {
                @Override
                public void run() {
                    lock.lock();
                    try {

```

```

        for (int i = 0; i < 20000; i++) {
            counter++;
        }
    } catch (Exception e) {
        // handle the exception
    } finally {
        lock.unlock();
    }
}
});
thread1.start();
thread2.start();

thread1.join();
thread2.join();
/* In ra kết quả cuối cùng của biến bên trong đối tượng */
System.out.println("Counter = " + counter);
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```