Họ và tên: Nguyễn Gia Bảo
Mã số sinh viên: 22520109
Lớp: IT007.O29.1

HỆ ĐIỀU HÀNH
BÁO CÁO LAB 5

CHECKLIST

5.5. BÀI TẬP THỰC HÀNH

	BT 1	BT 2	BT 3	BT 4
Trình bày cách làm	☒	☒	☒	☒
Chụp hình minh chứng	☒	☒	☒	☒
Giải thích kết quả	☒	☒	☒	☒

5.6. BÀI TẬP ÔN TẬP

	BT 1
Trình bày cách làm	☒
Chụp hình minh chứng	☒
Giải thích kết quả	☒

Tự chấm điểm:

**Lưu ý: Xuất báo cáo theo định dạng PDF, đặt tên theo cú pháp:*
<Tên nhóm>_LAB5.pdf

5.5. BÀI TẬP THỰC HÀNH

1. Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau: sells <= products <= sells + [4 số cuối của MSSV]

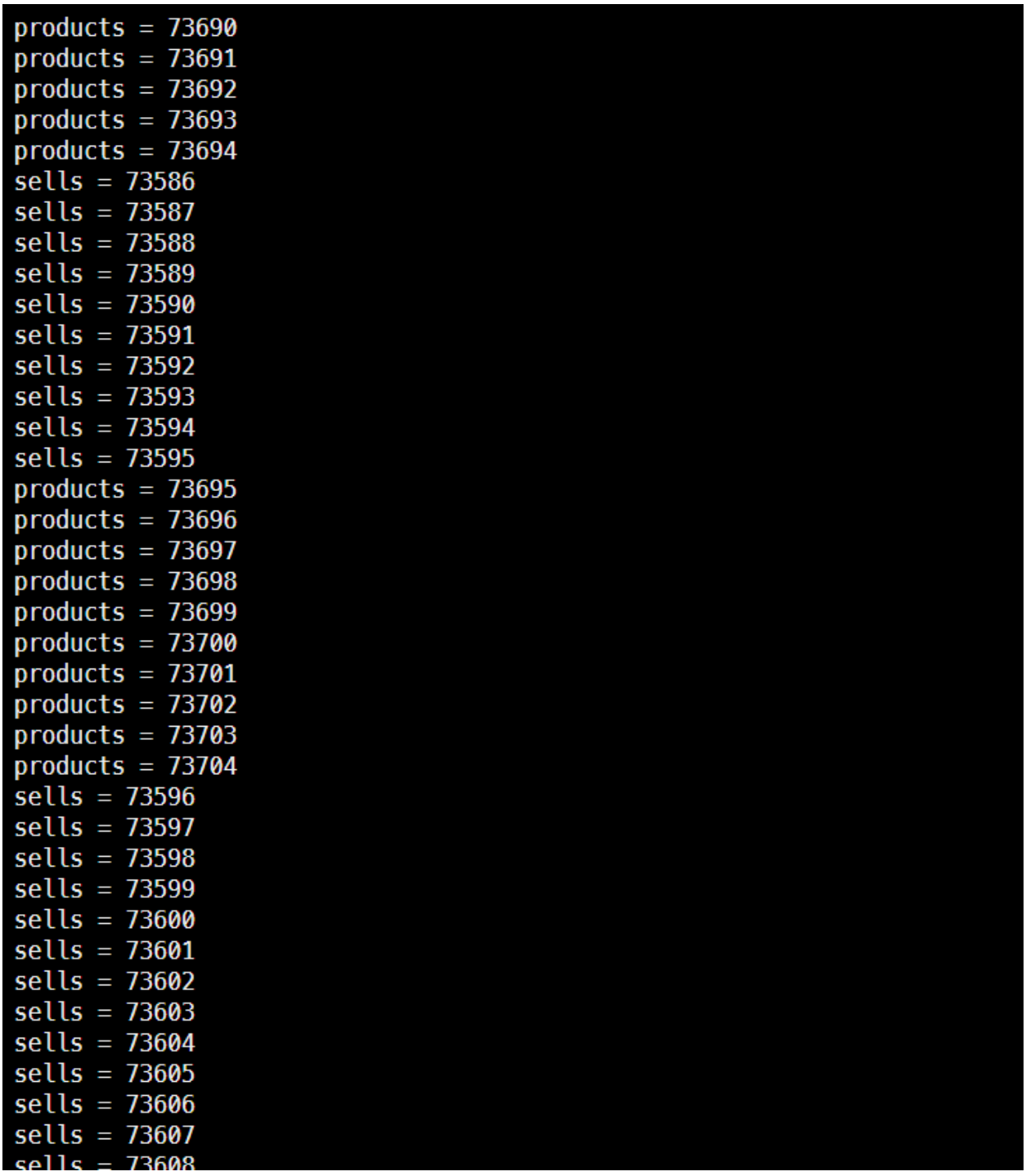
Trả lời...

- Ý tưởng thực hiện hiện: Dùng semaphore để đồng bộ 2 tiểu trình

bt1.c > product_add1(void *)

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

sem_t s1, s2;
int sells = 0;
int products = 0;

void *sell_add1(void *threadid){
    while(1){
        sem_wait(&s1);
        sells++;
        printf("sells = %d\n", sells);
        sem_post(&s2);
    }
}

void *product_add1(void *threadid){
    while(1){
        sem_wait(&s2);
        products++;
        printf("products = %d\n", products);
        sem_post(&s1);
    }
}

int main(){
    sem_init(&s1, 0, 0);
    sem_init(&s2, 0, 109);
    pthread_t idthread1, idthread2;
    pthread_create(&idthread1, NULL, sell_add1, NULL);
    pthread_create(&idthread2, NULL, product_add1, NULL);
    pthread_exit(NULL);
}
```

Code bài tập 1



Kết quả khi chạy

- Giải thích:

- Ta có 2 điều kiện : sells <= products và products <= sells + 109 (MSSV 22520109)

và ban đầu sells = products = 0 → để tăng sells thì products phải tăng trước và có thể tăng products 109 lần trước khi tăng sells.

- Khởi tạo semaphore s1 với giá trị ban đầu là 0 và semaphore s2 với giá trị ban đầu là 109 bằng sem_init().
- Đặt sem_wait(&s1) trước khi thực hiện sells++, sem_wait(&s2) trước khi thực hiện products++; sem_post(&s1) sau products++, sem_post(&s2) sau sells++.
- Sử dụng pthread_create() để tạo 2 tiểu trình thực hiện 2 công việc trên.

2. Cho một mảng a được khai báo như một mảng số nguyên có thể chứa n phần tử, a được khai báo như một biến toàn cục. Viết chương trình bao gồm 2 thread chạy song song:

- Một thread làm nhiệm vụ sinh ra một số nguyên ngẫu nhiên sau đó bỏ vào a. Sau đó đếm và xuất ra số phần tử của a có được ngay sau khi thêm vào.
- Thread còn lại lấy ra một phần tử trong a (phần tử bất kỳ, phụ thuộc vào người lập trình). Sau đó đếm và xuất ra số phần tử của a có được ngay sau khi lấy ra, nếu không có phần tử nào trong a thì xuất ra màn hình "Nothing in array a".

Chạy thử và tìm ra lỗi khi chạy chương trình trên khi chưa được đồng bộ. Thực hiện đồng bộ hóa với semaphore.

Trả lời...

- **Code trước khi đồng bộ bằng semaphore:**

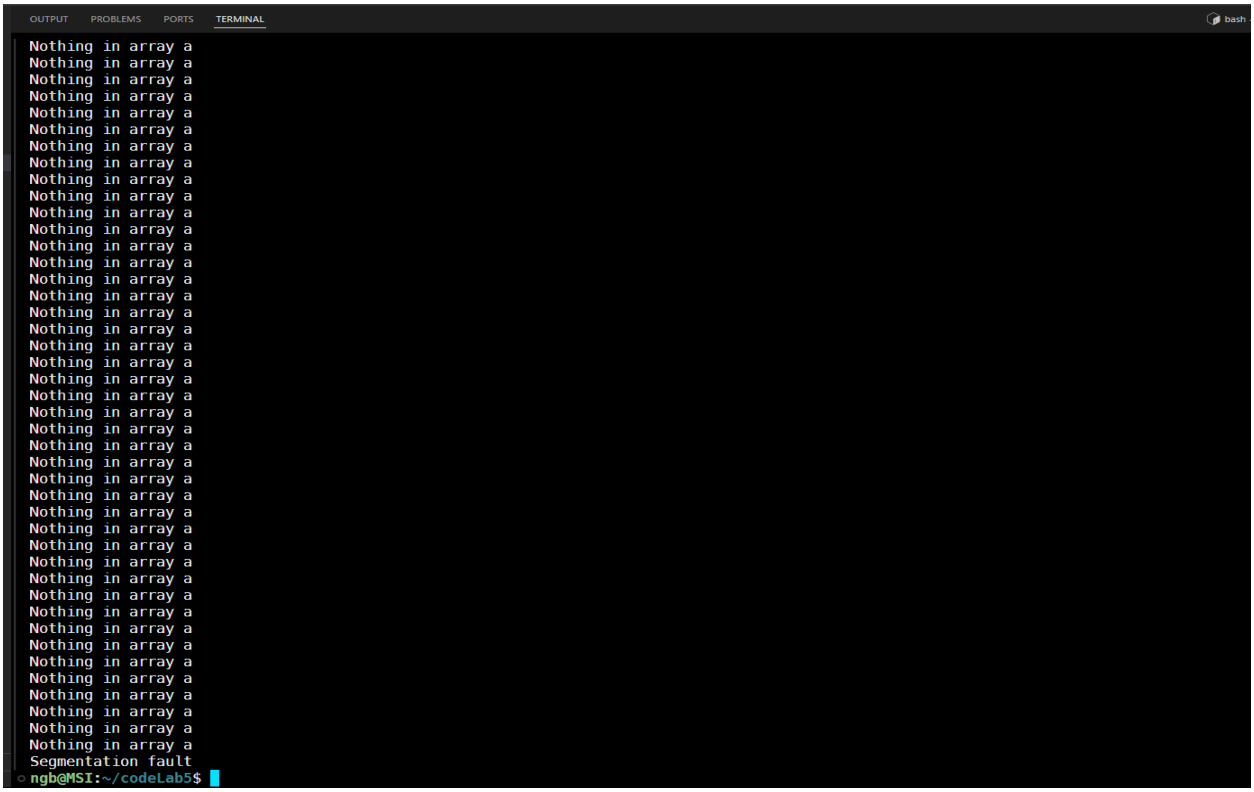
bt2.c > ...

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

int a[50];
int n = 0;

sem_t s;

void *add(void *threadid)
{
    while(1)
    {
        srand(time(NULL));
        int randomNumber = rand() % 100 + 1;

        //sem_wait(&s);

        a[n] = randomNumber;
        n++;
        printf("Add %d \n", randomNumber);
        printf("Remaining %d \n", n);

        //sem_post(&s);
        sleep(1);
    }
}
```

```c
void *remove_(void *threadid)
{
    while (1)
    {
        //sem_wait(&s);
        if (n == 0)
            printf("Nothing in array a\n");
        else
        {
            int rm = a[n - 1];
            n--;
            printf("Remove %d \n", rm);
            printf("Remaining %d \n", n);
        }
        //sem_post(&s);
        sleep(1);
    }
}

int main()
{
    //sem_init(&s, 0, 1);
    pthread_t idthread1, idthread2;

    pthread_create(&idthread1, NULL, add, NULL);
    pthread_create(&idthread2, NULL, remove_, NULL);

    pthread_join(idthread1, NULL);
    pthread_join(idthread2, NULL);
}

```

- **Thực thi code trước khi đồng bộ :**

Thực hiện khi có lệnh sleep(1) trong các hàm add và remove_



Thực hiện không có lệnh sleep(1) trong các hàm add và remove_

Code sau khi đồng bộ bằng semaphore:

- Khởi tạo một semaphore s, đặt lệnh sem_wait(&s) trước khi tiến trình thực hiện công việc và lệnh *sem_post(&s)* sau khi tiến trình hoàn thành công việc.

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

int a[10000];
int n = 0;

sem_t s;

void *add(void *threadid)
{
    while (1)
    {
        srand(time(NULL));
        int randomNumber = rand() % 100 + 1;

        sem_wait(&s);

        a[n] = randomNumber;
        n++;
        printf("Add %d \n", randomNumber);
        printf("Remaining %d \n", n);
        sem_post(&s);
    }
}
void *remove_(void *threadid)
```

```c
void *remove_(void *threadid)
{
    while (1)
    {
        sem_wait(&s);
        if (n == 0)
            printf("Nothing in array a\n");
        else
        {
            int rm = a[n - 1]; // xóa phần tử cuối của mảng a
            n--;
            printf("Remove %d \n", rm);
            printf("Remaining %d \n", n);
        }
        sem_post(&s);
    }
}

int main()
{
    sem_init(&s, 0, 1);
    pthread_t idthread1, idthread2;

    pthread_create(&idthread1, NULL, add, NULL);
    pthread_create(&idthread2, NULL, remove_, NULL);

    pthread_join(idthread1, NULL);
    pthread_join(idthread2, NULL);
}
```

- **Thực hiện code sau khi đã đồng bộ với semaphore:**

- **Giải thích:**

- Khi chạy chương trình chưa đồng bộ với semaphore, nếu không có lệnh sleep , chương trình sẽ dẫn đến race condition và dẫn đến lỗi
- Khi đồng bộ 2 chương trình với semaphore, chương trình sẽ đảm bảo chạy tốt và không có lỗi, vì lệnh *sem_wait(&s)* và *sem_post(&s)* giúp ngăn chặn việc các tiến trình tranh chấp, một tiến trình sẽ phải chờ đợi đến khi tiến trình kia hoàn tất công việc với các biến a và n rồi mới được thay đổi các biến trên.

3. Cho 2 process A và B chạy song song như sau:

int x = 0;	
PROCESS A	PROCESS B

processA()
{
while(1){
x = x + 1;
if (x == 20)
x = 0;
print(x);
}
}
	processB()
	{
	while(1){
	x = x + 1;
	if (x == 20)
	x = 0;
	print(x);
	}
	}

Hiện thực mô hình trên C trong hệ điều hành Linux và nhận xét kết quả.

Trả lời…

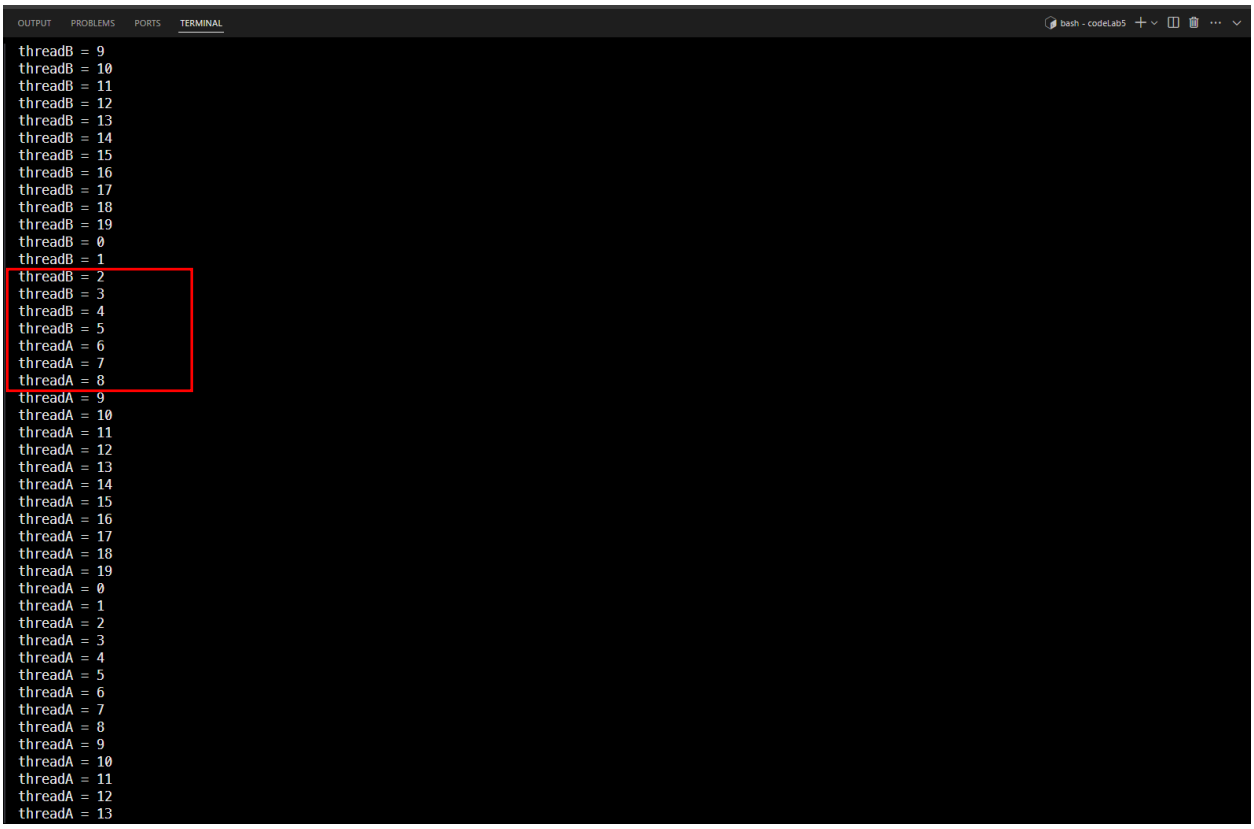
- **Code thực hiện yêu cầu của đề:**

bt1.c | bt2.c | bt3.c | bt2-semaphore.c | bt1

bt3.c > threadA(void *)

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

int x = 0;

void *threadA(void *threadid)
{
    while (1)
    {
        x = x + 1;
        if (x == 20)
        {
            x = 0;
        }
        printf("threadA = %d\n", x);
    }
}
void *threadB(void *threadid)
{
    while (1)
    {
        x = x + 1;
        if (x == 20)
        {
            x = 0;
        }
        printf("threadB = %d\n", x);
    }
}

int main()
{
    pthread_t idthreadA, idthreadB;
    pthread_create(&idthreadA, NULL, threadA, NULL);
    pthread_create(&idthreadB, NULL, threadB, NULL);
    pthread_exit(NULL);
}
```

- **Kết quả khi chạy chương trình trên:**



- **Nhận xét:**

- Kết quả in ra các giá trị bị ngắt quãng do 2 tiến trình đều thực hiện đồng thời và cùng tác động thay đổi giá trị biến x mà không có sự đồng bộ, dẫn đến việc các tiến trình in kết quả chồng lên nhau, không nhất quán từ 1 → 19 rồi lặp lại như ta muốn.

4. Đồng bộ với mutex để sửa lỗi bất hợp lý trong kết quả của mô hình Bài 3.

Trả lời…

- **Ý tưởng thực hiện:** dùng mutex để đồng bộ 2 tiểu trình A và B

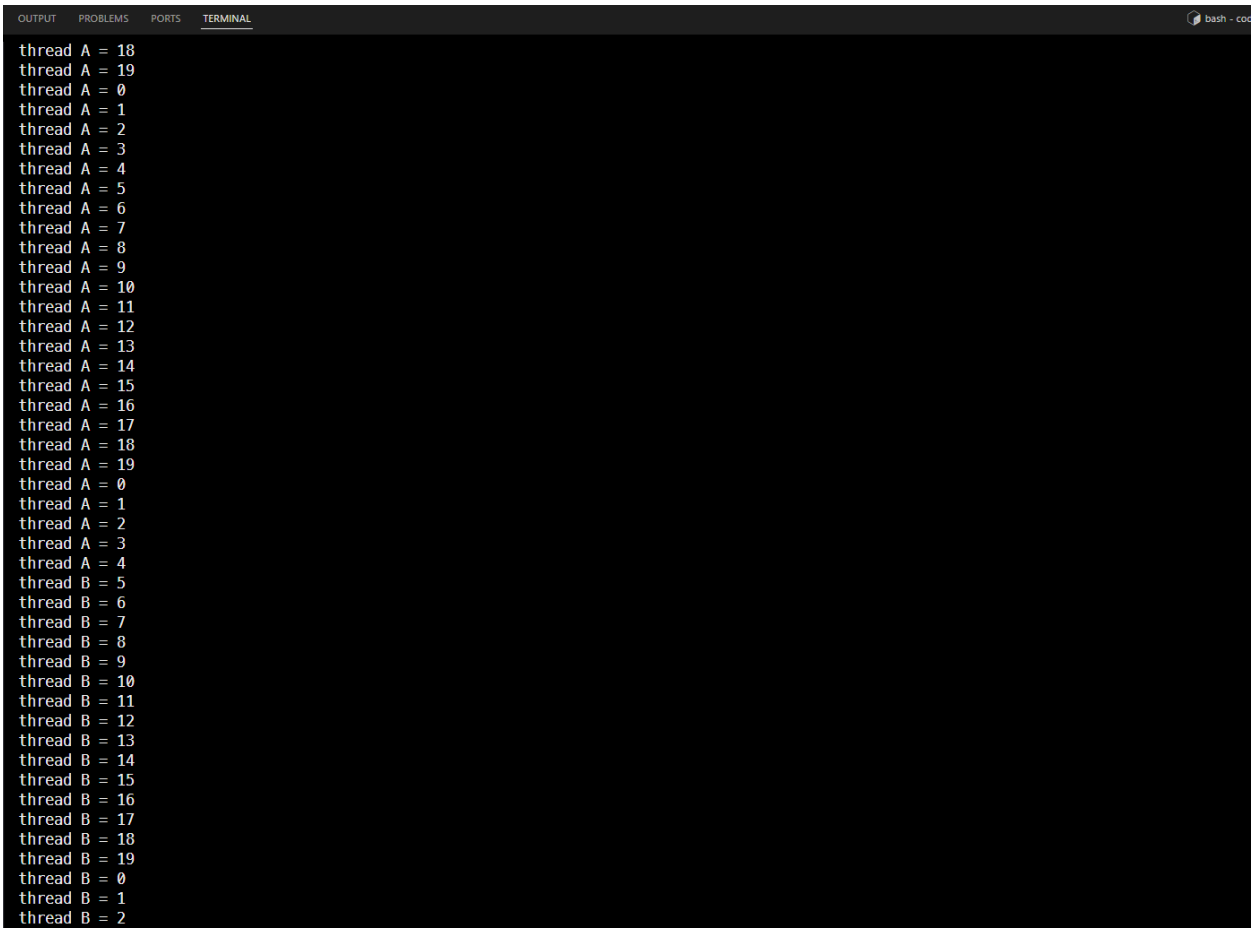
- **Code thực hiện :**

bt1.c | bt2.c | bt3.c | bt4.c | bt2-semaphore.c | bt1

bt4.c > threadA(void *)

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

int x = 0;
pthread_mutex_t m;

void *threadA(void *threadid)
{
    while (1)
    {
        pthread_mutex_lock(&m);
        x = x + 1;
        if (x == 20)
        {
            x = 0;
        }
        printf("thread A = %d\n", x);
        pthread_mutex_unlock(&m);
    }
}
void *threadB(void *threadid)
{
    while (1)
    {
        pthread_mutex_lock(&m);
        x = x + 1;
        if (x == 20)
        {
            x = 0;
        }
        printf("thread B = %d\n", x);
        pthread_mutex_unlock(&m);
    }
}

int main()
{
    pthread_mutex_init(&m, NULL);
    pthread_t idthreadA, idthreadB;
    pthread_create(&idthreadA, NULL, threadA, NULL);
    pthread_create(&idthreadB, NULL, threadB, NULL);
    pthread_exit(NULL);
}
```

- **Kết quả thực hiện:**



- **Giải thích:**

- Khai báo một mutex *m* , trước khi mỗi tiến trình thực hiện công việc, chúng ta sẽ khóa mutex lại bằng lệnh *"pthread_mutex_lock(&m)"* , khiến cho tiến trình còn lại không thể can thiệp thay đổi dữ liệu của biến x. Sau khi một tiến trình thực hiện xong, ta sẽ mở khóa mutex bằng lệnh *"pthread_mutex_unlock(&m)"*, để tiến trình còn lại có thể làm việc với biến x.

- Sau khi áp dụng mutex, kết quả của biến x đã lần lượt được in ra từ 0 → 19 rồi lặp lại.

5.6. BÀI TẬP ÔN TẬP

1. **Biến ans được tính từ các biến x1, x2, x3, x4, x5, x6 như sau:**

w = x1 * x2; (a)

v = x3 * x4; (b)

y = v * x5; (c)

z = v * x6; (d)

y = w * y; (e)

z = w * z; (f)

ans = y + z; (g)

Giả sử các lệnh từ (a) → (g) nằm trên các thread chạy song song với nhau. Hãy lập trình mô phỏng và đồng bộ trên C trong hệ điều hành Linux theo thứ tự sau:

- (c), (d) chỉ được thực hiện sau khi v được tính
- (e) chỉ được thực hiện sau khi w và y được tính
- (g) chỉ được thực hiện sau khi y và z được tính

Trả lời...

- **Ý tưởng:** Dùng semaphore để đồng bộ các tiểu trình
- **Code thực hiện:**

bt1.c | bt2.c | bt3.c | bt4.c | bt5.c | bt2-semaphore.c | bt1

bt5.c > ...

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

int w, v, y, z, ans;
int x1 = 2, x2 = 3, x3 = 1, x4 = 5, x5 = 4, x6 = 3;

sem_t s1, s2, s3, s4, s5;

void *a(void *threadid)
{
    printf("a :\n");
    w = x1 * x2;
    sem_post(&s4);
}

void *b(void *threadid)
{
    printf("b :\n");
    v = x3 * x4;
    sem_post(&s1);
    sem_post(&s1);
}

void *c(void *threadid)
{
    sem_wait(&s1);
    printf("c :\n");
    y = v * x5;
    sem_post(&s2);
}

void *d(void *threadid)
{
    sem_wait(&s1);
    printf("d :\n");
    z = v * x6;
}
```

```c

void *e(void *threadid)
{
    sem_wait(&s2);
    sem_wait(&s4);
    printf("e :\n");
    y = w * y;
    sem_post(&s3);
}

void *f(void *threadid)
{
    printf("f :\n");
    z = w * z;
    sem_post(&s5);
}

void *g(void *threadid)
{
    sem_wait(&s3);
    sem_wait(&s5);
    printf("g :\n");

    ans = y + z;
}
```

```
64
65  int main()
66  {
67      sem_init(&s1, 0, 0);
68      sem_init(&s2, 0, 0);
69      sem_init(&s3, 0, 0);
70      sem_init(&s4, 0, 0);
71      sem_init(&s5, 0, 0);
72
73      pthread_t idthread1, idthread2, idthread3, idthread4, idthread5, idthread6, idthread7;
74
75      pthread_create(&idthread1, NULL, a, NULL);
76      pthread_create(&idthread2, NULL, b, NULL);
77      pthread_create(&idthread3, NULL, c, NULL);
78      pthread_create(&idthread4, NULL, d, NULL);
79      pthread_create(&idthread5, NULL, e, NULL);
80      pthread_create(&idthread6, NULL, f, NULL);
81      pthread_create(&idthread7, NULL, g, NULL);
82
83      pthread_join(idthread1, NULL);
84      pthread_join(idthread2, NULL);
85      pthread_join(idthread3, NULL);
86      pthread_join(idthread4, NULL);
87      pthread_join(idthread5, NULL);
88      pthread_join(idthread6, NULL);
89      pthread_join(idthread7, NULL);
90  }
```

- **Kết quả thực hiện:**

```
OUTPUT  PROBLEMS  PORTS  TERMINAL                                  bash - c
ngb@MSI:~/codeLab5$ gcc -o bt5 bt5.c
ngb@MSI:~/codeLab5$ ./bt5
a :
b :
c :
d :
e :
f :
g :
ngb@MSI:~/codeLab5$
```

Thực hiện các tiểu trình theo thứ tự

- **Giải thích:**

- Tạo 5 semaphore s1, s2, s3, s4, s5 với giá trị khởi đầu là 0 (có thể sử dụng mutex).

- Để (c), (d) chỉ được thực hiện sau khi v được tính ta đặt sem_wait(&s1) trước (c), (d) và 2 lần sem_post(&s1) sau (b).

- Để (e) chỉ được thực hiện sau khi w và y được tính ta đặt sem_wait(&s2) và sem_wait(&s4) trước (e), sem_post(&s2) và sem_post(&s4) sau (c), (a).

- Để (g) chỉ được thực hiện sau khi y và z được tính ta đặt sem_wait(&s3) và sem_wait(&s5) trước (g), sem_post(&s3) và sem_post(&s5) sau (e), (f).

- Sử dụng pthread_create() để tạo 7 tiểu trình thực hiện 7 công việc.
- Sử dụng pthread_join() để đảm bảo rằng tất cả tiểu trình đều đã thực hiện xong công việc trước khi thoát chương trình.