Họ và tên: Bùi Lê Nhật Tri

MSSV: 23521634

Lớp: IT00007.P11.1

BÁO CÁO LAB 5

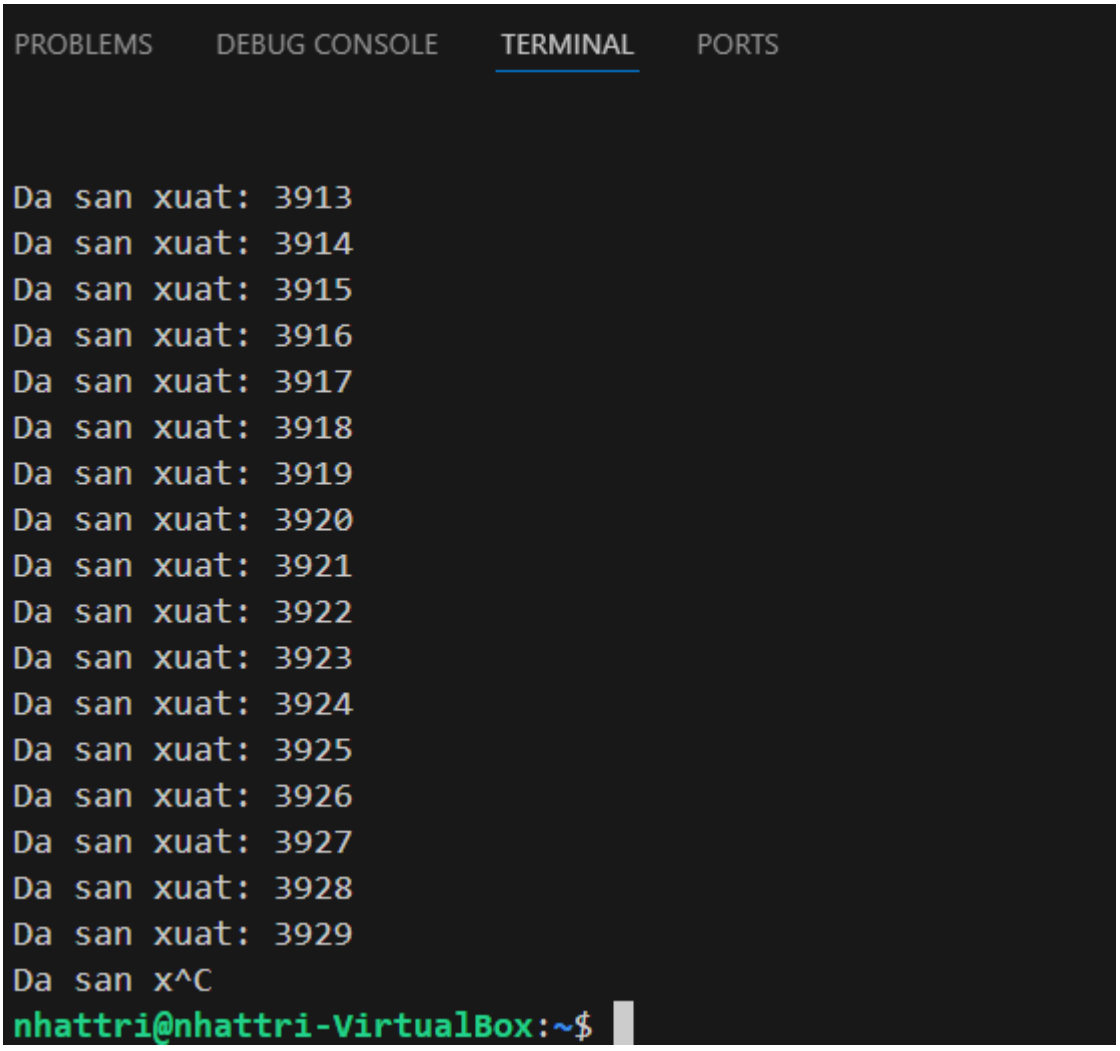
Câu 1: Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau: sells <= products <= sells + [4 số cuối của MSSV]

Code:

```cpp
#include <pthread.h>
#include <semaphore.h>

using namespace std;

#define MSSV 1634 // mssv La 23521634

int products = 0;
int sells = 0;

sem_t sem;

void* producer(void* arg) {
    while (true) {
        sem_wait(&sem);
        if (products <= sells + MSSV) {
            products++;
            printf("Da san xuat: %d\n", products);
        }
        sem_post(&sem);
    }
    return NULL;
}

void* consumer(void* arg) {
    while (true) {
        sem_wait(&sem);
        if (products >= sells) {
            sells++;
            printf("Da tieu thu: %d\n", sells);
        }
        sem_post(&sem);
    }
}

int main() {
    pthread_t prod_thread, con_thread;

    sem_init(&sem, 0, 1);

    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&con_thread, NULL, consumer, NULL);

    pthread_join(prod_thread, NULL);
    pthread_join(con_thread, NULL);

    sem_destroy(&sem);

    return 0;
}
```

Hình 1: Source code câu 1

Kết quả:



Hình 2:Kết quả câu 1

Giải thích:

- **Producer** chỉ sản xuất khi chưa đạt giới hạn (không vượt quá sells + 1634).
- **Consumer** chỉ tiêu thụ khi có ít nhất một sản phẩm (products > sells).
- **Semaphore** đảm bảo rằng không có luồng nào thay đổi giá trị của sells và products cùng lúc, tránh race condition và đảm bảo các điều kiện đồng bộ được thỏa mãn.

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.

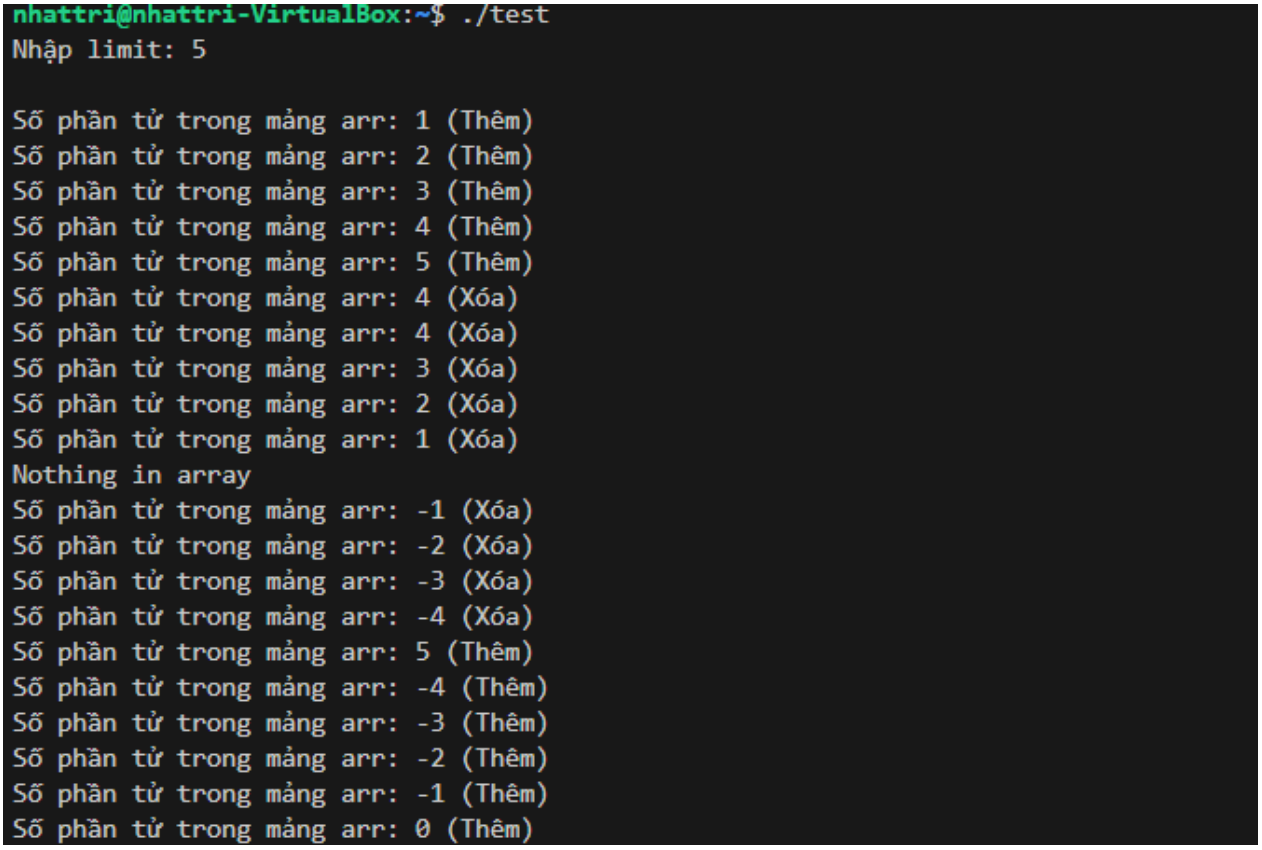
Code không có semaphore:

Code:

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

int arr[10];
int counter = 0, limit;

void *ProcessA(void *arg) {
    while (1) {
        if (counter < limit) {
            arr[counter++] = rand() % 100 + 1;
            printf("\nSố phần tử trong mảng arr: %d (Thêm)", counter);
        }
    }
    return NULL;
}

void *ProcessB(void *arg) {
    while (1) {
        for (int i = 0; i < counter; i++) {
            arr[i] = arr[i + 1];
        }
        counter--;

        if (counter == 0) {
            printf("\nNothing in array");
        } else if (counter < limit) {
            printf("\nSố phần tử trong mảng arr: %d (Xóa)", counter);
        }
    }
    return NULL;
}

int main() {
    printf("Nhập limit: ");
    scanf("%d", &limit);

    pthread_t pA, pB;
    pthread_create(&pA, NULL, ProcessA, NULL);
    pthread_create(&pB, NULL, ProcessB, NULL);

    pthread_join(pA, NULL);
    pthread_join(pB, NULL);

    return 0;
}
```

Hình 3:Source code câu 2a

Kết quả:



Hình 4:Kết quả câu 2a

Giải thích:

- Tiểu Trình A không có phần tử nào để xử lý nhưng Tiểu Trình B vẫn tiếp tục thay đổi giá trị của counter, điều này có thể do thiếu đồng bộ hóa, khiến luồng B có thể thay đổi giá trị mà không có sự kiểm soát từ luồng A.

Code có semaphore:

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>
#include <semaphore.h>
int arr[10];
int counter = 0, limit;
sem_t sem_add, sem_remove;

void *ProcessA(void *arg) {
    while (1) {
        sem_wait(&sem_add);
        if (counter < limit) {
            arr[counter++] = rand() % 100 + 1;
            printf("\nSố phần tử trong mảng arr sau khi thêm: %d", counter);
        }
        sem_post(&sem_remove);
    }
    return NULL;
}

void *ProcessB(void *arg) {
    while (1) {
        sem_wait(&sem_remove);
        if (counter > 0) {
            for (int i = 0; i < counter - 1; i++) {
                arr[i] = arr[i + 1];
            }
            counter--;

            if (counter == 0) {
                printf("\nNothing in array");
            } else if (counter < limit) {
                printf("\nSố phần tử trong mảng arr sau khi xóa: %d", counter);
            }
        }
        sem_post(&sem_add);
    }
    return NULL;
}
```

Hình 5:Source code câu 2b.1

```c
int main() {
    printf("Nhập limit: ");
    scanf("%d", &limit);

    sem_init(&sem_add, 0, 1);
    sem_init(&sem_remove, 0, 0);

    pthread_t pA, pB;
    pthread_create(&pA, NULL, ProcessA, NULL);
    pthread_create(&pB, NULL, ProcessB, NULL);

    pthread_join(pA, NULL);
    pthread_join(pB, NULL);

    sem_destroy(&sem_add);
    sem_destroy(&sem_remove);

    return 0;
}
```

Hình 6: Source code câu 2b.2

Kết quả:

```
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử trong mảng arr sau khi thêm: 1
Nothing in array
Số phần tử ^C
nhattri@nhattri-VirtualBox:~$
```

Hình 7:Kết quả câu 2b

Giải thích:

- Khi sử dụng 2 semaphore là sem_add và sem_remove để kiểm soát 2 tiểu trình thêm và xóa. Chúng ta có thể thấy số phần tử của mảng a luôn dương.

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

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.

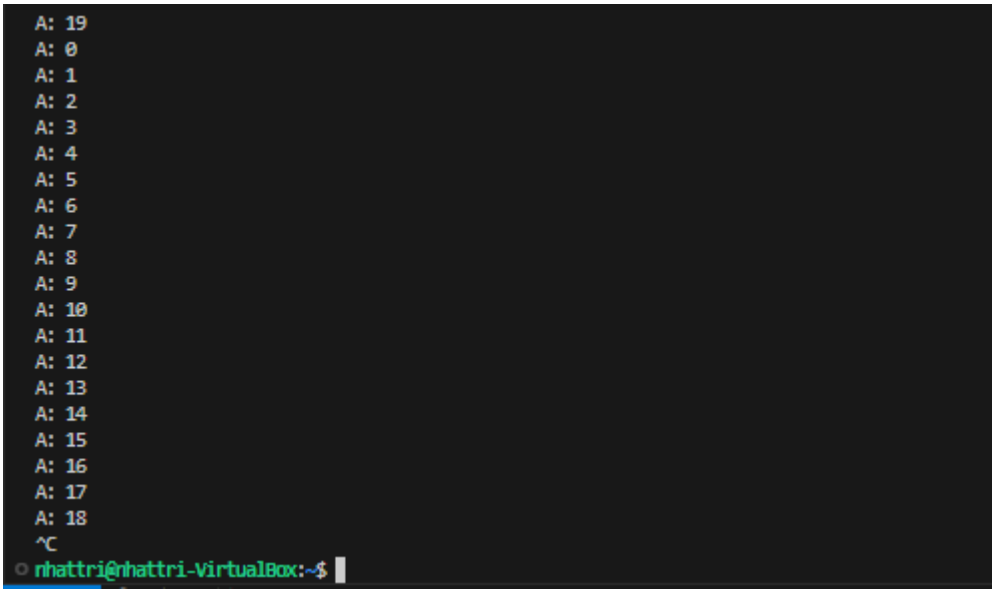
Code:

test.cpp > ProcessB(void *)

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

int x = 0;

void *ProcessA(void *arg)
{
    while (1)
    {
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("A: %d\n", x);
    }
    return NULL;
}

void *ProcessB(void *arg)
{
    while (1)
    {
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("B: %d\n", x);
    }
    return NULL;
}

int main()
{
    pthread_t pA, pB;

    pthread_create(&pA, NULL, ProcessA, NULL);
    pthread_create(&pB, NULL, ProcessB, NULL);

    pthread_join(pA, NULL);
    pthread_join(pB, NULL);

    return 0;
}
```

Hình 8:Source code câu 3

Kết quả:



Hình 9:Kết quả câu 3

Giải thích:

- **Không chính xác**: Giá trị của x không thể chắc chắn theo đúng thứ tự, vì không có bảo vệ đồng bộ. Chẳng hạn, cả hai luồng có thể cùng đọc giá trị x và tăng nó, dẫn đến việc mất dữ liệu. Điều này tạo ra một hành vi không xác định và khó đoán.
- **In kết quả sai**: Bạn có thể thấy kết quả không đồng nhất, với các giá trị x không liên tục và không như mong đợi. Điều này xảy ra vì các luồng thay đổi x đồng thời mà không có cơ chế đồng bộ.

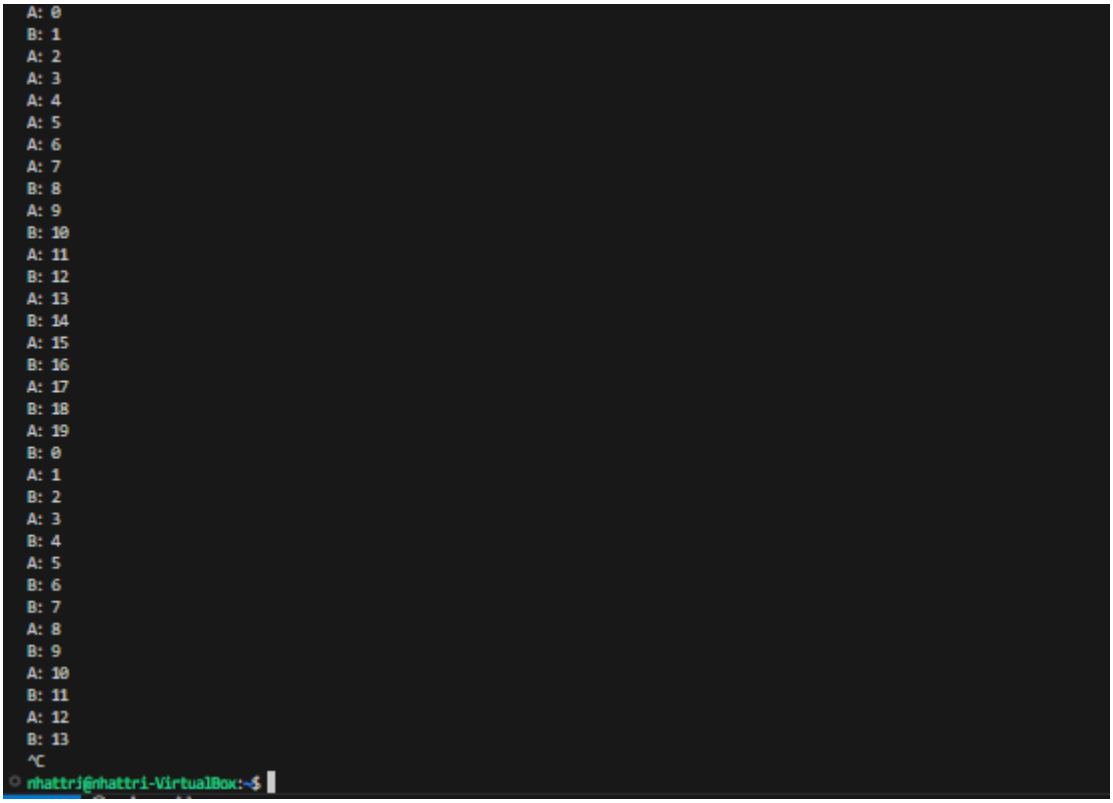
4. Đồng bộ với mutex để sửa lỗi bất hợp lý trong kết quả của mô hình Bài 3.

Code:

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

int x = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *ProcessA(void *arg)
{
    while (1)
    {
        pthread_mutex_lock(&mutex);
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("A: %d\n", x);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void *ProcessB(void *arg)
{
    while (1)
    {
        pthread_mutex_lock(&mutex);
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("B: %d\n", x);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main()
{
    pthread_t pA, pB;

    pthread_create(&pA, NULL, ProcessA, NULL);
    pthread_create(&pB, NULL, ProcessB, NULL);

    pthread_join(pA, NULL);
    pthread_join(pB, NULL);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Hình 10:Source code câu 4

Kết quả:



Hình 11:Kết quả câu 4

Giải thích:

- **pthread_mutex_lock(&mutex)**: Khóa mutex trước khi thay đổi biến x. Điều này đảm bảo rằng chỉ có một luồng có thể thay đổi giá trị của x tại một thời điểm.
- **pthread_mutex_unlock(&mutex)**: Giải phóng mutex sau khi thay đổi xong. Sau khi mutex được giải phóng, luồng khác có thể tiếp tục thay đổi giá trị của x.
- **pthread_mutex_destroy(&mutex)**: Hủy mutex sau khi sử dụng xong, để giải phóng tài nguyên.