Name: NGUYỄN ANH TÀI - ID: 20520924

Class: IT007.M22.1

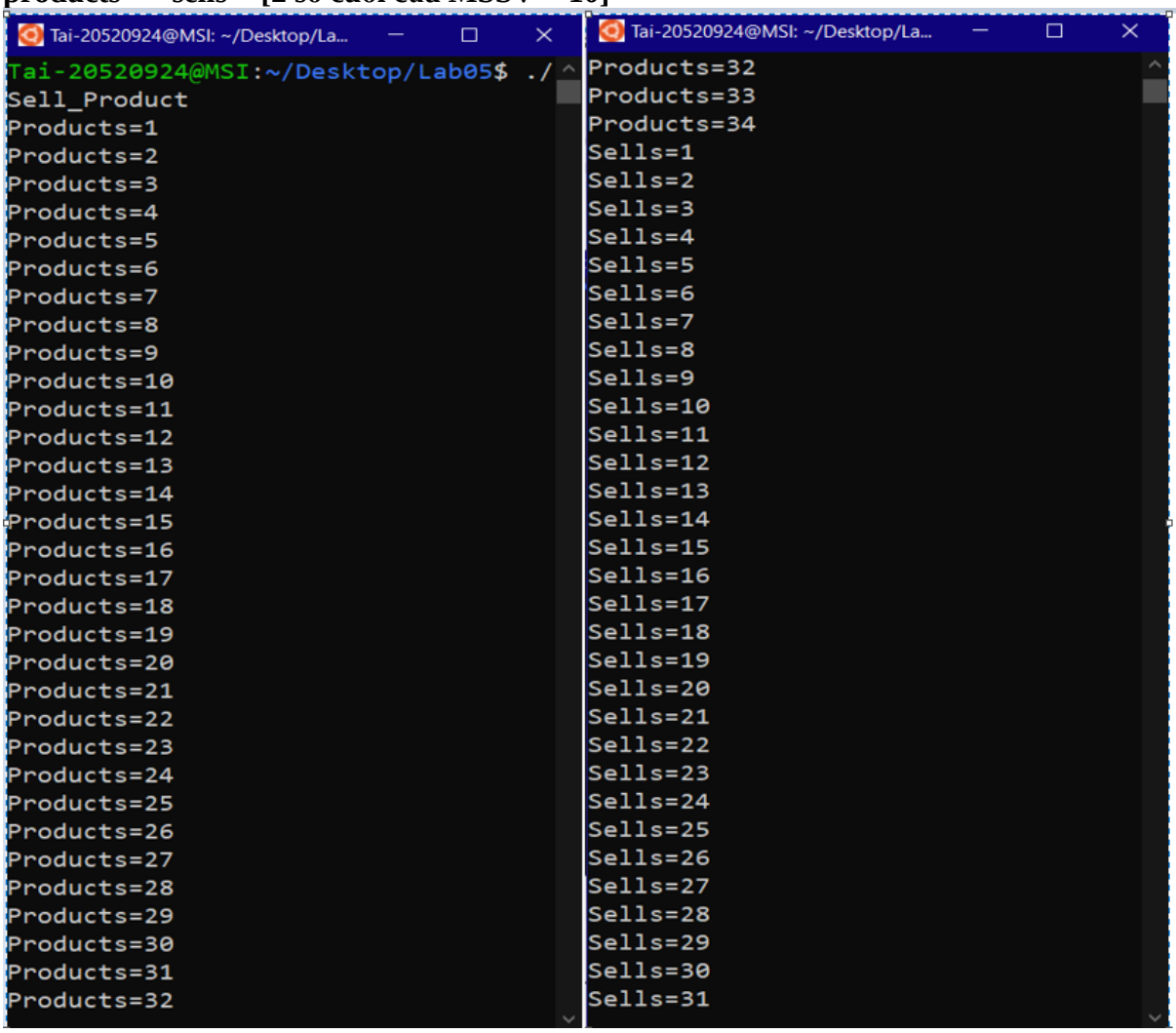
OPERATING SYSTEM
LAB 05'S REPORT

SUMMARY

Task	Status
1. Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau: sells <= products <= sells + [2 số cuối của MSSV + 10]	DONE
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.	DONE
3. Hiện thực mô hình bài 3 trên C trong hệ điều hành Linux và nhận xét kết quả.	DONE
4. Đồng bộ với mutex để sửa lỗi bất hợp lý trong kết quả của mô hình Bài 3	Done
5. …	Undone

Self-scrores: 8

TASK 01: Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau: sells <= products <= sells + [2 số cuối của MSSV + 10]



Hình 1:Sau khi products tăng và bị giới hạn lại, sau đó Sells mới được tăng

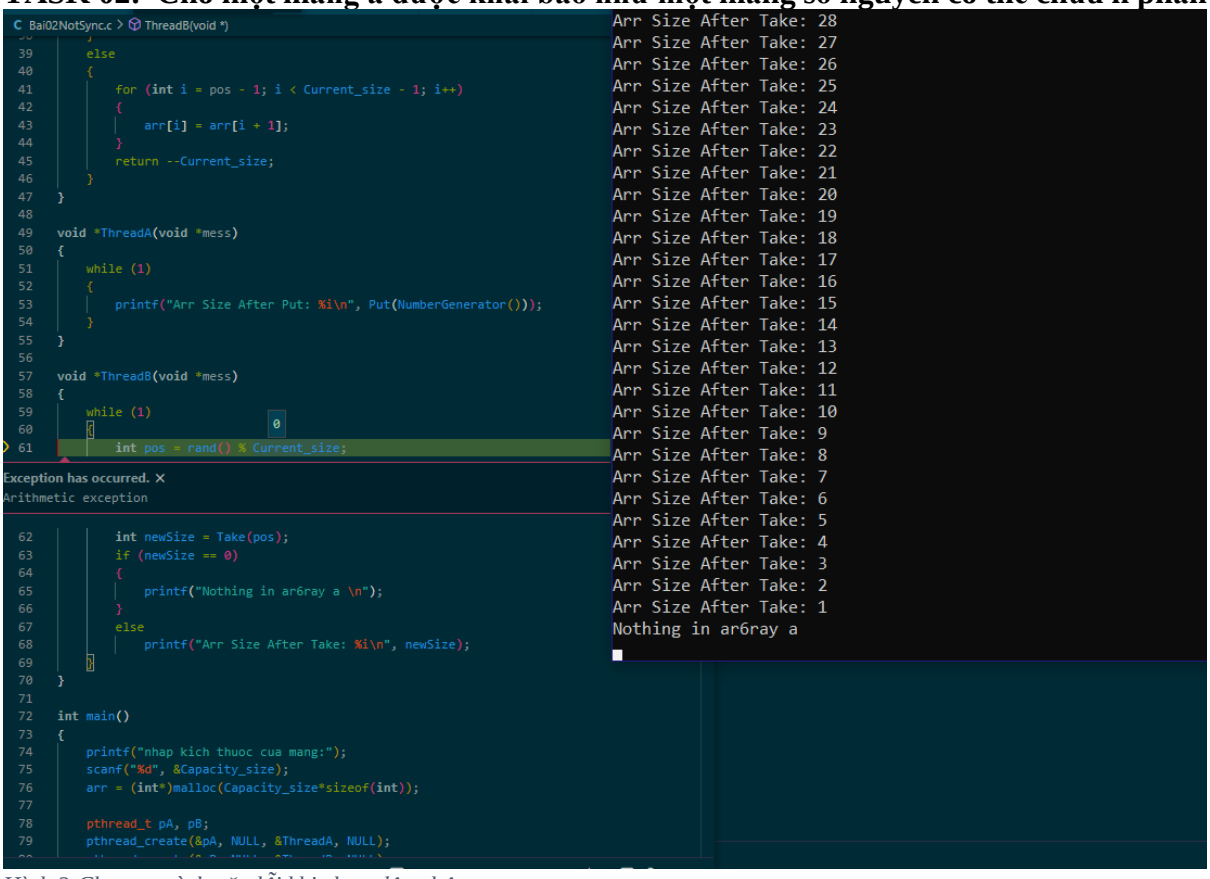
Code:

```c
/*################
#University of Information Technology
#IT007 Operating System
#Nguyen Anh Tai - 20520924
#File:Bai01.c
################*/
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
int sells = 0, products;
sem_t sem1, sem2;
void *ProcessA(void *mess)
{
```

```
    while (1)
    {
        sem_wait(&sem1);
        sells++;
        printf("Sells=%d\n", sells);
        sem_post(&sem2);
    }
}
void *ProcessB(void *mess)
{
    while (1)
    {
        sem_wait(&sem2);
        products++;
        printf("Products=%d\n", products);
        sem_post(&sem1);
    }
}
int main()
{
    sem_init(&sem1, 0, 0);
    sem_init(&sem2, 0, 34);
    pthread_t pA, pB;
    pthread_create(&pA, NULL, &ProcessA, NULL);
    pthread_create(&pB, NULL, &ProcessB, NULL);
    while (1)
    {
    }
    return 0;
}
```

Giải Thích:
sem1 đảm bảo rằng (sells<=products) (khởi tạo = 0),
sem2 đảm bảo rằng (products <= sells + 34)(khởi tạo = 34)
Khi mới bắt đầu tiến trình A sẽ bị chặn lại bởi lệnh sem_wait(&sem1).
Khi ProcessB chạy. Products++ và sem2 -- (bởi sem_wait(&sem2)) và sem1++ sau mỗi vòng lặp cho đến khi sem2=0.
Sau đó ProcessA được chạy (khi giá trị của sem1 > 0).
Khi đó Sells++ và sem2++ (bởi hàm sem_post(&sem2)) và sem1-- (bởi sem_wait(&sem1)) cho đến khi sem1=0.
Hai Tiến trình lúc này sẽ luân phiên nhau tăng product và sell đồng thời cũng sẽ đảm bảo được điều kiện đề bài cho **sells <= products <= sells + [2 số cuối của MSSV + 10]**

TASK 02: Cho một mảng a được khai báo như một mảng số nguyên có thể chứa n phần tử…



Hình 2:Chương trình gặp lỗi khi chưa động bộ.

Code:

```
/*################
#University of Information Technology
#IT007 Operating System
#Nguyen Anh Tai - 20520924
#File:Bai02NotSync.c
################*/
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>

int *arr;
int Capacity_size = 0;
int Current_size = 0;

int Put(int value)
```

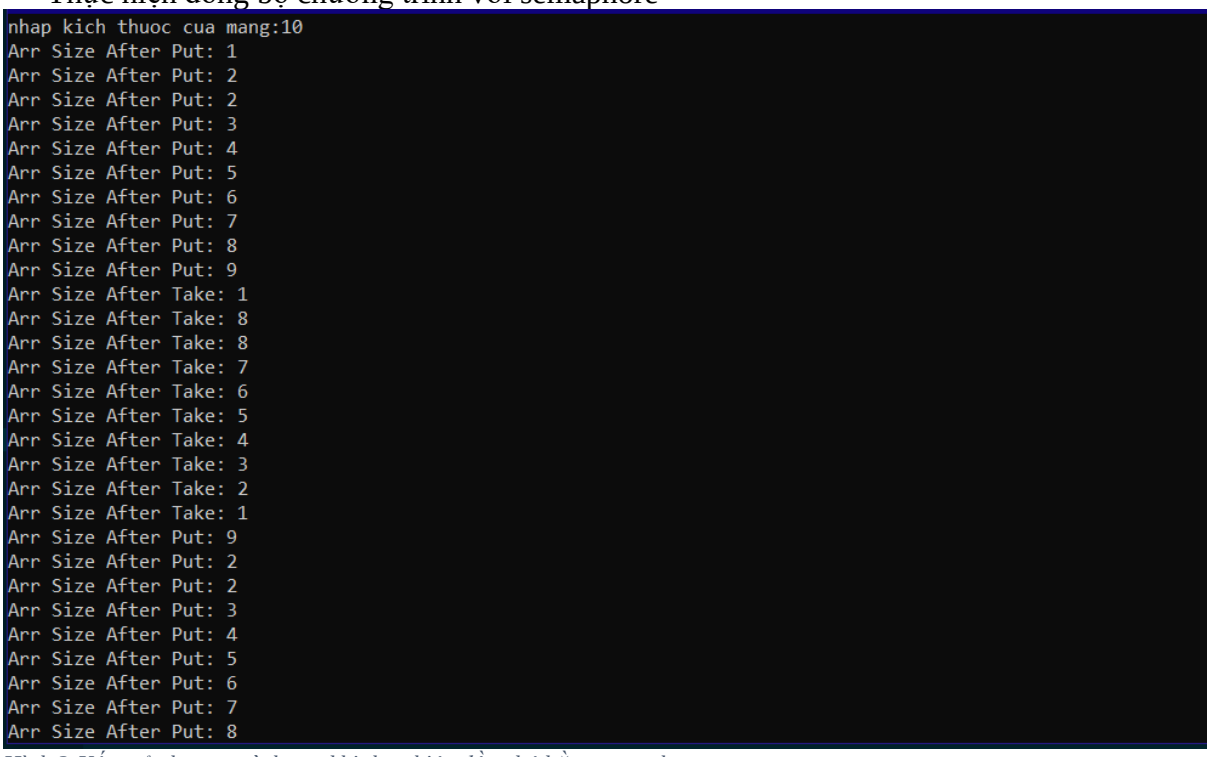
```
{
    arr[Current_size] = value;
    return ++Current_size;
}

int NumberGenerator()
{
    srand(time(NULL));
    return rand();
}

int Take(int pos)
{
    if (pos == 0)
    {
        for (int i = pos; i < Current_size - 1; i++)
        {
            arr[i] = arr[i + 1];
        }
        return --Current_size;
    }
    else
    {
        for (int i = pos - 1; i < Current_size - 1; i++)
        {
            arr[i] = arr[i + 1];
        }
        return --Current_size;
    }
}

void *ThreadA(void *mess)
{
    while (1)
    {
        printf("Arr Size After Put: %i\n", Put(NumberGenerator()));
    }
}

void *ThreadB(void *mess)
{
    while (1)
    {
        int pos = rand() % Current_size;
        int newSize = Take(pos);
```

```
        if (newSize == 0)
        {
            printf("Nothing in ar6ray a \n");
        }
        else
            printf("Arr Size After Take: %i\n", newSize);
    }
}

int main()
{
    printf("nhap kich thuoc cua mang:");
    scanf("%d", &Capacity_size);
    arr = (int*)malloc(Capacity_size*sizeof(int));

    pthread_t pA, pB;
    pthread_create(&pA, NULL, &ThreadA, NULL);
    pthread_create(&pB, NULL, &ThreadB, NULL);
    while (1)
    {
    }
    return 0;
}
```

Giải thích:

Thread A thêm phần tử vào mảng vào không kịp, trong khi đó Thread B lấy phân tử khỏi mảng khi mà Arr không có phần tử nào, điều đó gây ra lỗi.

=> Thực hiện đồng bộ chương trình với semaphore



Hình 3:Kết quả chương trình sau khi thực hiện đồng bộ bằng semaphore

Code:

```c
/*################
#University of Information Technology
#IT007 Operating System
#Nguyen Anh Tai - 20520924
#File:Bai02Sync.c
################*/
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>

int *arr;
int Capacity_size = 0;
int Current_size = 0;
sem_t semPut, semTake;

int Put(int value)
{

    arr[Current_size] = value;
    return ++Current_size;
```

```
}

int NumberGenerator()
{
    srand(time(NULL));
    return rand();
}

int Take(int pos)
{
    if (pos == 0)
    {
        for (int i = pos; i < Current_size - 1; i++)
        {
            arr[i] = arr[i + 1];
        }
        return --Current_size;
    }
    else
    {
        for (int i = pos - 1; i < Current_size - 1; i++)
        {
            arr[i] = arr[i + 1];
        }
        return --Current_size;
    }
}

void *ThreadPut(void *mess)
{
    while (1)
    {
        sem_wait(&semTake);
        printf("Arr Size After Put: %i\n", Put(NumberGenerator()));
         sem_post(&semPut);
    }
}

void *ThreadTake(void *mess)
{
    while (1)
    {
        sem_wait(&semPut);
        int pos = rand() % Current_size;
        int newSize = Take(pos);
```

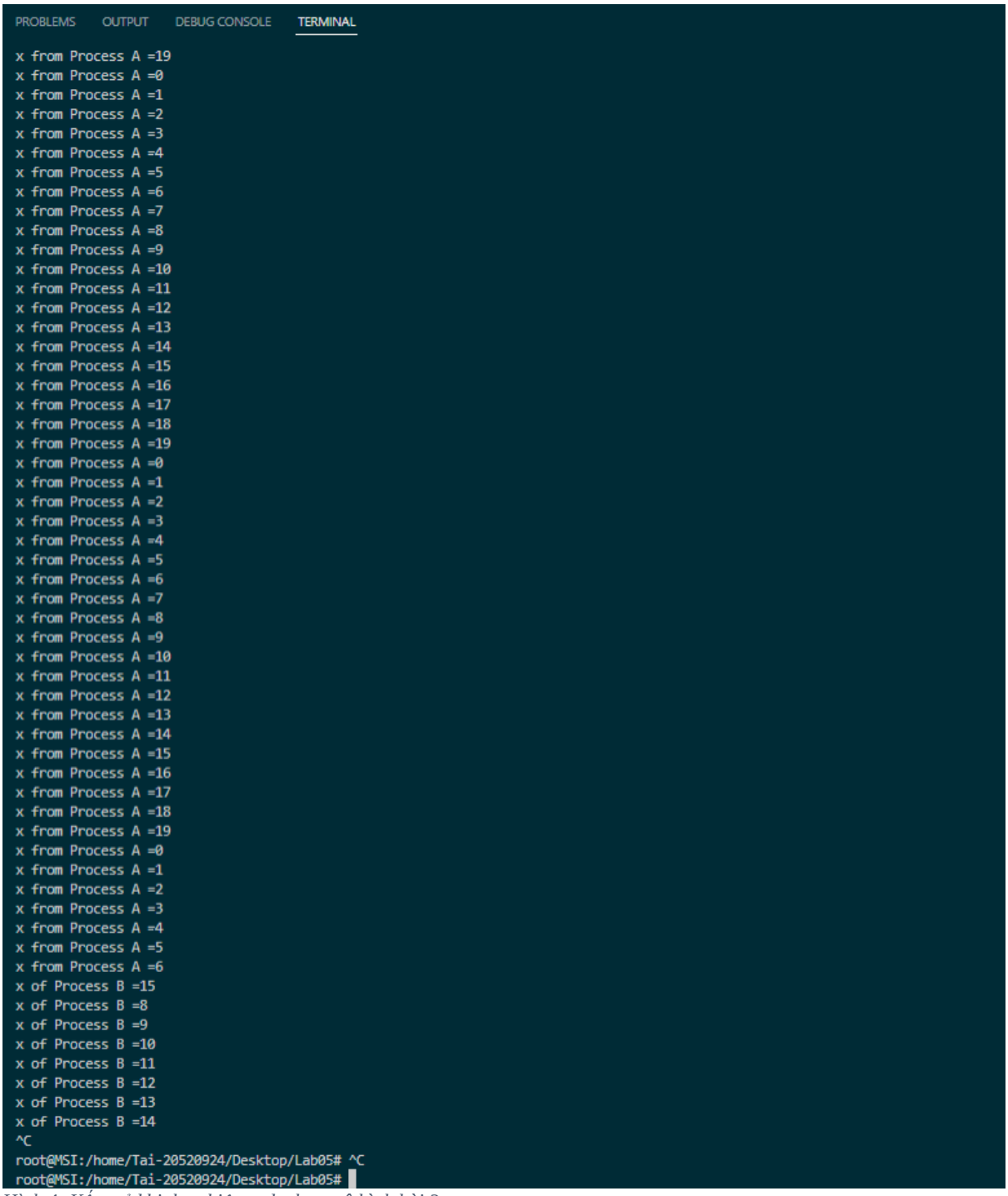
```
        if (newSize == 0)
        {
            printf("Nothing in ar6ray a \n");
        }
        else
        {
            printf("Arr Size After Take: %i\n", newSize);
        }
        sem_post(&semTake);
    }
}

int main()
{
    printf("nhap kich thuoc cua mang:");
    scanf("%d", &Capacity_size);
    arr = (int *)malloc(Capacity_size * sizeof(int));
    sem_init(&semTake, 0, Capacity_size);
    sem_init(&semPut, 0, 0);
    pthread_t pA, pB;

    pthread_create(&pB, NULL, & ThreadTake, NULL);
    pthread_create(&pA, NULL, & ThreadPut, NULL);
    while (1)
    {
    }
    return 0;
}
```

Giải thích:
Ta tạo 2 semaphore mang tên SemPut (khởi tạo = 0) và SemTake(khởi tạo =Capacity_size), ở 2 tiến trình đề bài yêu cầu, gọi là ThreadTake và ThreadPut:
+ khi ThreadTake muốn lấy một phần tử ở mảng Arr thì ThreadTake sẽ bị chặn lại bởi Sem_wait(&SemPut) tức là khi trong mảng Arr chưa có phần tử nào thì ThreadTake sẽ phải đợi tới khi có phần tử. Điều này cũng có nghĩa là semtake sẽ đóng vai trò đảm bảo rằng không có hành động lấy phần tử khỏi mảng Arr khi mà mảng này không có phần tử.
+ Ở ThreadPut khi muốn đặt một phần tử vào thì cũng sẽ bị chặn bởi Sem_wait(&SemTake), ThreadPut sẽ không thể đặt thêm phần tử vào khi mà Array có số phần tử bằng Capacity_size, Vậy SemPut cũng sẽ đảm bảo được rằng sẽ không có phần tử nào được thêm vào mảng Arr khi mảng đang đầy.

TASK 03: Hiện thực mô hình bài 3 trong hệ điều hành Linux và nhận xét kết quả.

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

```c
int x = 0;

void *ProcessA(void *mess)
{
    while (1)
    {
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("x from Process A =%d\n", x);
    }
}

void *ProcessB(void *mess)
{
    while (1)
    {
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("x of Process B =%d\n", x);
    }
}
int main()
{
    pthread_t pA, pB;
    pthread_create(&pA, NULL, &ProcessA, NULL);
    pthread_create(&pB, NULL, &ProcessB, NULL);
    while (1)
    {
    }
    return 0;
}
```



Hình 4: Kết quả khi thực hiện code theo mô hình bài 3

Kết quả của mô hình ở bài 3 là kết quả sai do hiện tượng tại cùng một thời điểm khi mà Process A đang tính toán trên biến x thì Process B lại xen lấy dữ liệu của biến x để tính toán, điều này dẫn đến khi process A thực hiện tính toán xong trước thì Process B lại lấy kết quả đầu ra của biến A trước đó để đem đi tính toán

Vậy nếu ta thực hiện code theo mô hình của bài 3 thì sẽ gây là việc không nhất quán về dữ liệu.

TASK 04: Đồng bộ với mutex để sửa lỗi bất hợp lý trong kết quả của mô hình Bài 3.

```
x from Process A =17
x from Process A =18
x from Process A =19
x from Process A =0
x from Process A =1
x from Process A =2
x from Process A =3
x from Process A =4
x from Process A =5
x from Process A =6
x from Process A =7
x from Process A =8
x from Process A =9
x from Process A =10
x from Process A =11
x from Process A =12
x from Process A =13
x from Process A =14
x from Process A =15
x from Process A =16
x from Process A =17
x from Process A =18
x from Process A =19
x from Process A =0
x from Process A =1
x from Process A =2
x from Process A =3
x of Process B =4
x of Process B =5
x of Process B =6
x of Process B =7
x of Process B =8
x of Process B =9
x of Process B =10
x of Process B =11
x of Process B =12
x of Process B =13
x of Process B =14
x of Process B =15
x of Process B =16
x of Process B =17
x of Process B =18
x from Process A =19
x from Process A =0
x from Process A =1
```

Hình 5: Kết quả sau khi đã mô hình 3 được đồng bộ hoá bằng mutex

Code:

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

#include <pthread.h>

int x = 0;
pthread_mutex_t mutex;

void *ProcessA(void *mess)
{
    while (1)
```

```
    {
        pthread_mutex_lock(&mutex);
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("x from Process A =%d\n", x);
        pthread_mutex_unlock(&mutex);
    }
}

void *ProcessB(void *mess)
{
    while (1)
    {
        pthread_mutex_lock(&mutex);
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("x of Process B =%d\n", x);
        pthread_mutex_unlock(&mutex);
    }
}
int main()
{
    pthread_t pA, pB;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&pA, NULL, &ProcessA, NULL);
    pthread_create(&pB, NULL, &ProcessB, NULL);
    while (1)
    {
    }
    return 0;
}
```

Giải Thích:

ta đặt mutex_lock và mutex_unlock ở trước và sau khi thực hiện việc tính toán ở 2 process, điều này sẽ đảm bảo không có tiến trình nào có thể lấy được dữ liệu của tiến trình khác khi nó đang thực thi.