Name: Nguyễn Đại Hưng

ID: 24520601

Class: IT007.Q15.1

OPERATING SYSTEM
LAB 5'S REPORT

SUMMARY

Self-scrores: 10

Note: Export file to **PDF and name the file by following format:*
Student ID_LABx.pdf

Section 5.5

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].**

4 số cuối MSSV là 0601 ➔ sells <= products <= sells + 0601.

```c
/*#######################################
# University of Information Technology   #
# IT007 - Operating Systems              #
# Nguyen Dai Hung - 24520601             #
# File: Bai001.c                         #
#######################################*/

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

int sells = 0;
int products = 0;
sem_t sem_empty;  // Kiểm tra còn chỗ sản xuất không (products - sells < 601)
sem_t sem_full;   // Kiểm tra có hàng để bán không (products > sells)

void* ProcessA(void* mess)
{
    while(1)
    {
        sem_wait(&sem_full);
        sells++;
        printf("A sells: sells=%d, products=%d, diff=%d\n",
               sells, products, products - sells);
        sem_post(&sem_empty);
    }
}

void* ProcessB(void* mess)
{
    while(1)
    {
        sem_wait(&sem_empty);
        products++;
        printf("B produces: sells=%d, products=%d, diff=%d\n",
               sells, products, products - sells);
        sem_post(&sem_full);
    }
}

int main()
{
    int fourLastNumberInMyStudentID = 601;

    sem_init(&sem_full, 0, 0);                          // Ban đầu chưa có hàng
    sem_init(&sem_empty, 0, fourLastNumberInMyStudentID);   // Ban đầu có thể sản xuất 601 sản phẩm

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

    while(1)
    {
    }

    return 0;
}
```

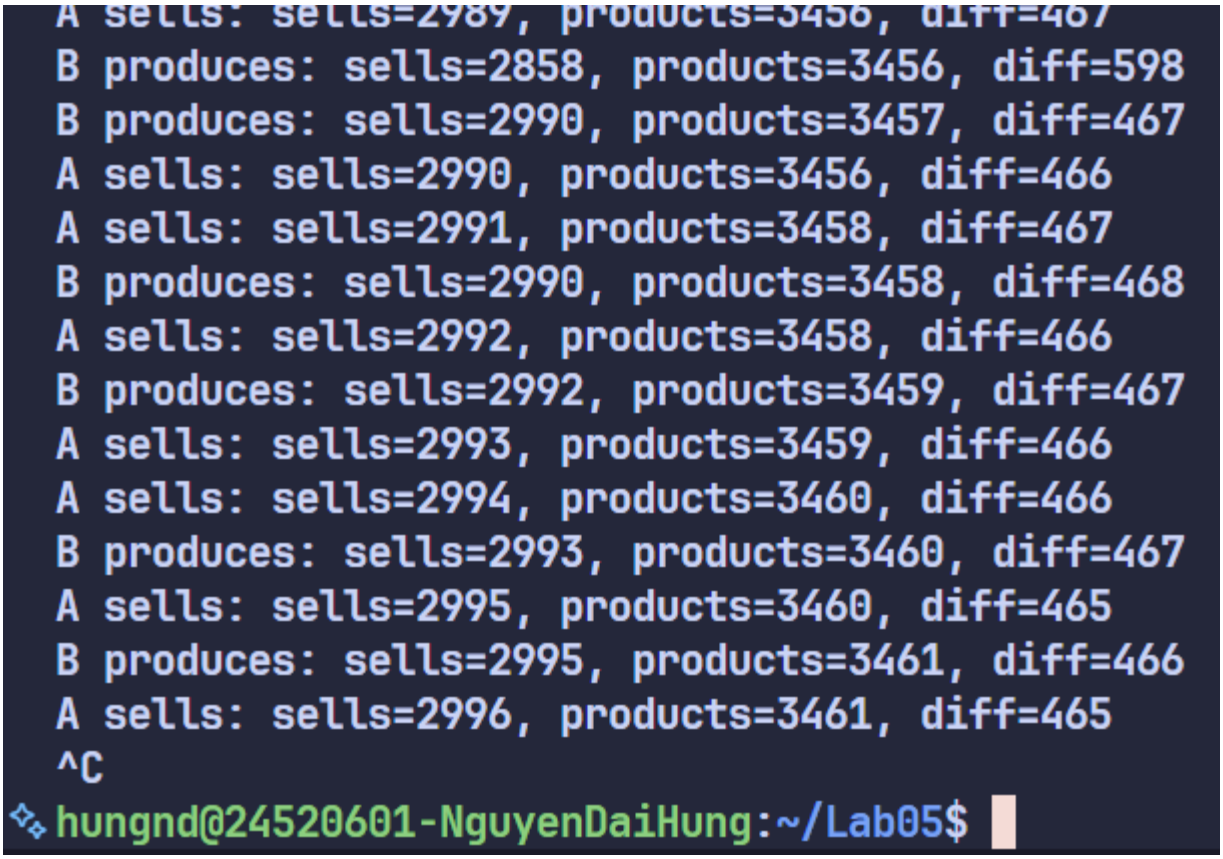
Hình 1: Source code bài 001.

Giải thích code:

- *Bài toán yêu cầu viết chương trình mô phỏng quá trình sản xuất và bán hàng với điều kiện: sells ≤ products ≤ sells + 601 (Nghĩa là số sản phẩm đã bán không được vượt quá số sản phẩm đã sản xuất, đồng thời số sản phẩm tồn kho không được vượt quá 601 (4 số cuối MSSV).*
- *Bài toán có 2 ràng buộc chính:*
    - ***Không bán khi chưa có hàng**: Process A phải đợi Process B sản xuất.*

- ***Không sản xuất quá 601 sản phẩm tồn kho****: Process B phải dừng khi kho đầy.*

- *Để giải quyết, ta sử dụng 2 semaphore:*
  - ***sem_full****: Đếm số hàng có sẵn (khởi tạo = 0).*
  - ***sem_empty****: Đếm số chỗ trống trong kho (khởi tạo = 601).*
- *Cách thức hoạt động:*
  - ***Process A (Bán hàng):***
    1. *Gọi sem_wait(&sem_full) – Kiểm tra có hàng không?*
       - *Nếu sem_full = 0: bị chặn, đợi có hàng.*
       - *Nếu sem_full > 0: giảm 1 và tiếp tục.*
    2. *Tăng sell++ – Bán 1 sản phẩm.*
    3. *Gọi sem_post(&sem_empty) – Báo có thêm chỗ trống.*
  - ***Process B (Sản xuất):***
    1. *Gọi sem_wait(&sem_empty) – Kiểm tra còn chỗ không?*
       - *Nếu sem_empty = 0: bị chặn, đợi có chỗ trống.*
       - *Nếu sem_empty > 0: giảm 1 và tiếp tục.*
    2. *Tăng product++ – Sản xuất 1 sản phẩm.*
    3. *Gọi sem_post(&sem_full) – Báo có hàng mới.*



Hình 2: Kết quả chạy bài 001.

Nhận xét kết quả:

- *Giá trị diff = products - sells luôn nằm trong khoảng [0, 601], chứng tỏ:*
    - *Không bao giờ bán khi chưa có hàng (diff ≥ 0).*
    - *Không bao giờ sản xuất quá 601 sản phẩm tồn kho (diff ≤ 601).*
- *Semaphore đảm bảo đồng bộ vì:*
    - ***sem_wait()** sẽ chặn thread nếu semaphore = 0, tránh vi phạm điều kiện.*
    - ***sem_post()** đánh thức thread đang bị chặn, cho phép tiếp tục.*
    - *Hai semaphore hoạt động như "phiếu lấy hàng" và "phiếu sản xuất", tổng luôn bằng 601.*

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.

a) Chạy chương trình khi chưa được đồng bộ.

```c
/*#######################################
# University of Information Technology   #
# IT007 - Operating Systems              #
# Nguyen Dai Hung - 24520601             #
# File: Ba1002_RaceCondition.c           #
#######################################*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

#define MAX_SIZE 10

int a[MAX_SIZE];
int size = 0;

void PrintArray(int arr[], int current_size)
{
    if (current_size == 0)
    {
        printf("Nothing in array a.\n");
        return;
    }

    printf("[");
    for (int i = 0; i < current_size; i++)
    {
        printf("%d", arr[i]);
        if (i < current_size - 1)
            printf(", ");
    }
    printf("]\n");
}

void PrintDetailsOfArray(int arr[], int current_size, const char* label)
{
    printf("--- %s ---\n", label);
    printf("Number of elements in array: %d/%d\n", current_size, MAX_SIZE);
    printf("Elements in array: ");
    PrintArray(arr, current_size);
    printf("---------------------------\n\n");
}

void* ProcessA(void* arr)
{
    while (1)
    {
        if (size < MAX_SIZE)
        {
            int temp = rand() % 1000;

            a[size] = temp;
            usleep(1);
            size++;

            printf(">>> Process A (Producer) added %d to the array.\n", temp);
            PrintDetailsOfArray(a, size, "Producer Added");
        }
        else
        {
            printf("Producer: Buffer is FULL. Waiting...\n");
        }
    }
    return NULL;
}

void* ProcessB(void* arr)
{
    while (1)
    {
        if (size > 0)
        {
            int removedElement = a[size - 1];
            usleep(1);
            size--;

            printf("<<< Process B (Consumer) removed %d from the array.\n", removedElement);

        }

        PrintDetailsOfArray(a, size, "Consumer Removed/Attempt");
    }
    return NULL;
}

int main()
{
    srand(time(NULL));

    printf("Starting Producer-Consumer simulation (UNSYNCHRONIZED) with MAX_SIZE = %d\n", MAX_SIZE);

    pthread_t pA, pB;

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

    while (1)
    {
        sleep(1);
    }

    return 0;
}
```

Hình 3: Source code bài 002a.

Giải thích code:

- *Đây là một chương trình Producer-Consumer (Sản xuất - Tiêu thụ) cơ bản sử dụng đa luồng (pthreads) trong C, nhưng không hề có cơ chế đồng bộ hóa (như mutex hay semaphore).*
- *Các thành phần chính:*
    - *Mảng chia sẻ:*
        - *int arr[MAX_SIZE]: Mảng chứa các phần tử (buffer), có kích thước tối đa là MAX_SIZE = 10.*
        - *int current_size = 0: Biến toàn cục đếm số phần tử hiện có trong mảng.*
        - ***Lưu ý:** Cả arr và current_size là các tài nguyên chia sẻ được truy cập bởi cả hai luồng.*
    - *Hàm printDetailsOfArray:*
        - *In ra thông tin chi tiết của mảng: số lượng phần tử hiện tại (current_size) và các phần tử trong mảng.*
    - *Hàm ProcessA (Producer - Sản xuất):*
        - *Điều kiện đầy: Kiểm tra nếu mảng chưa đầy (current_size < MAX_SIZE).*
        - *Nếu chưa đầy:*
            - *Tạo số ngẫu nhiên (temp = rand() % 1000).*
            - *Thao tác thêm: gán arr[current_size] = temp; và tăng current_size lên 1.*
            - *In thông báo "Producer Added" và chi tiết mảng.*
        - *Nếu đầy: in thông báo "Buffer is FULL. Waiting...".*
        - *Luồng tạm dừng 1 giây (sleep(1)).*
    - *Hàm ProcessB (Consumer - Tiêu thụ):*
        - *Điều kiện Rỗng: Kiểm tra nếu mảng còn phần tử (current_size > 0).*
        - *Nếu còn phần tử:*
            - *Thao tác lấy: Lấy phần tử cuối cùng arr[current_size - 1], giảm current_size xuống 1. (Khu vực Găng - Critical Section)*
            - *In thông báo "Consumer Removed" và chi tiết mảng.*
        - *Nếu rỗng: In thông báo "Nothing in array a".*
        - *Luồng tạm dừng 1 giây (sleep(1)).*
    - *Hàm main:*
        - *Khởi tạo trình tạo số ngẫu nhiên.*

        - *Tạo hai luồng: pA (chạy ProcessA) và pB (chạy ProcessB).*
        - *Chương trình chính tạm dừng 30 giây để mô phỏng.*
- *Vấn đề cốt lõi: KHÔNG ĐỒNG BỘ HÓA*
    - *Các thao tác cập nhật mảng (arr và current_size) trong ProcessA và ProcessB là các Khu vực Găng (Critical Section). Khi hai luồng truy cập và sửa đổi các biến này cùng lúc mà không có cơ chế khóa (như mutex), sẽ xảy ra Điều kiện Tranh chấp (Race Condition), dẫn đến mất dữ liệu và trạng thái không nhất quán của mảng.*

```
--------------------------

>>> Process A (Producer) added 984 to the array.
<<< Process B (Consumer) removed 716 from the array.
--- Consumer Removed/Attempt ---
Number of elements in array: 2/10
Elements in array: [372, 716]
--------------------------

--- Producer Added ---
Number of elements in array: 2/10
Elements in array: [372, 716]
--------------------------

<<< Process B (Consumer) removed 716 from the array.
>>> Process A (Producer) added 57 to the array.
--- Producer Added ---
Number of elements in array: 2/10
Elements in array: [372, 716]
--------------------------

--- Consumer Removed/Attempt ---
Number of elements in array: 2/10
Elements in array: [372, 716]
--------------------------

>>> Process A (Producer) added 435 to the array.
--- Producer Added ---
```

Hình 4: Kết quả chạy bài 002a.

Nhận xét kết quả:

- *Kết quả chạy của chương trình chưa được đồng bộ hóa. Chúng ta có thể thấy rõ ràng lỗi Điều kiện Tranh chấp (Race Condition) và trạng thái mảng không nhất quán.*
- *Lỗi xảy ra chủ yếu do cơ chế đọc/ghi biến current_size non-atomic.*

- *Kết luận: Việc không sử dụng Mutex hoặc các cơ chế đồng bộ khác dẫn đến Điều kiện Tranh chấp (Race Condition). Các luồng đọc và ghi vào biến chia sẻ current_size và mảng arr một cách không kiểm soát, gây ra:*
    - *Lỗi cập nhật bị mất (Lost Update): Giá trị của current_size không phản ánh đúng số phần tử thực tế.*
    - *Lỗi ghi đè dữ liệu (Data Corruption): Producer có thể ghi vào vị trí mà Consumer vừa lấy ra (hoặc ngược lại), làm cho các phần tử trong mảng bị sai lệch so với thao tác mong muốn.*

b) Chạy chương trình khi đồng bộ hóa với senaphore.

```c
/*########################################
# University of Information Technology   #
# IT007 - Operating Systems              #
# Nguyen Dai Hong - 24520481              #
# File: Bai002_Semaphore.c               #
########################################*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>

#define MAX_SIZE 10

int a[MAX_SIZE];
int size = 0;

sem_t empty_slots;
sem_t full_slots;
pthread_mutex_t mutex;

void PrintArray(int arr[], int current_size)
{
    if (current_size == 0)
    {
        printf("Nothing in array a.\n");
        return;
    }

    printf("[");
    for (int i = 0; i < current_size; i++)
    {
        printf("%d", arr[i]);
        if (i < current_size - 1)
            printf(", ");
    }
    printf("]\n");
}

void PrintDetailsOfArray(int arr[], int current_size, const char* label)
{
    printf("--- %s ---\n", label);
    printf("Number of elements in array: %d/%d\n", current_size, MAX_SIZE);
    printf("Elements in array: ");
    PrintArray(arr, current_size);
    printf("--------------------------\n\n");
}

void* Process4(void* arr)
{
    while (1)
    {
        int temp = rand() % 1000;

        sem_wait(&empty_slots);
        pthread_mutex_lock(&mutex);

        a[size] = temp;
        size++;

        printf(">>> Process A (Producer) added %d to the array.\n", temp);
        PrintDetailsOfArray(a, size, "Producer Added");

        pthread_mutex_unlock(&mutex);
        sem_post(&full_slots);

        usleep(100000);
    }
    return NULL;
}

void* Process8(void* arr)
{
    while (1)
    {
        sem_wait(&full_slots);
        pthread_mutex_lock(&mutex);
        int removedElement = a[size - 1];
        size--;

        printf("<<< Process B (Consumer) removed %d from the array.\n", removedElement);
        PrintDetailsOfArray(a, size, "Consumer Removed");

        pthread_mutex_unlock(&mutex);
        sem_post(&empty_slots);

        usleep(150000);
    }
    return NULL;
}

int main()
{
    srand(time(NULL));

    printf("Starting Producer-Consumer simulation (SYNCHRONIZED) with MAX_SIZE = %d\n", MAX_SIZE);

    pthread_t pA, pB;

    sem_init(&empty_slots, 0, MAX_SIZE);
    sem_init(&full_slots, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&pA, NULL, Process4, NULL);
    pthread_create(&pB, NULL, Process8, NULL);

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

    sem_destroy(&empty_slots);
    sem_destroy(&full_slots);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

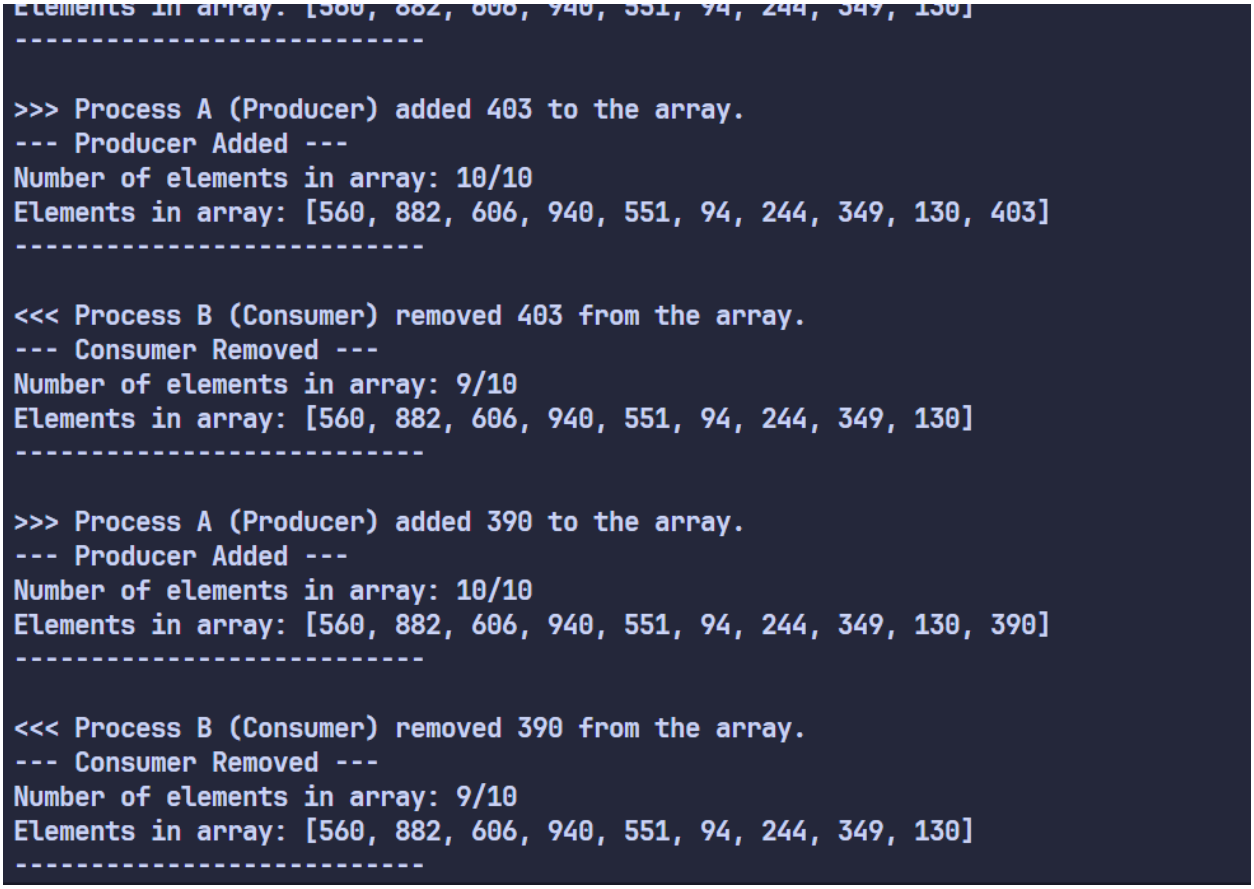
Hình 5: Source code bài 002b.

Giải thích code:

- *Đây là một chương trình hiện thực hóa cổ điển của bài toán **Producer-Consumer** sử dụng hai cơ chế đồng bộ hóa: **Semaphore** (cho điều kiện giới hạn) và **Mutex** (cho truy cập tài nguyên chia sẻ).*
- *Các thành phần đồng bộ:*
    - *sem_empty_slots (Semaphore):*
        - *Khởi tạo: MAX_SIZE = 10.*
        - *Ý nghĩa: Đếm số lượng chỗ trống có sẵn trong mảng (buffer).*
        - *Sử dụng: Producer gọi sem_wait() trên semaphore này. Nếu count=0 (Buffer đầy), Producer sẽ bị chặn, đảm bảo không sản xuất quá giới hạn.*
    - *sem_full_slots (Semaphore):*
        - *Khởi tạo: 0.*
        - *Ý nghĩa: Đếm số lượng phần tử đã lấp đầy (sản phẩm có sẵn) trong mảng.*
        - *Sử dụng: Consumer gọi sem_wait() trên semaphore này. Nếu count = 0 (Buffer rỗng), Consumer sẽ bị chặn, đảm bảo không tiêu thụ khi không có hàng.*
    - *gMutex (Mutex):*
        - *Khởi tạo: Khóa mở.*
        - *Ý nghĩa: Cơ chế khóa tương hỗ loại trừ (Mutual Exclusion Lock).*
        - *Sử dụng: Đặt xung quanh Critical Section - các đoạn mã truy cập và sửa đổi biến chia sẻ (a và size)—để đảm bảo chỉ một luồng có thể thực thi các thao tác này tại một thời điểm.*
- *Hoạt động của ProcessA (Producer - Sản xuất):*
    1. *sem_wait(&sem_empty_slots): Đợi chỗ trống.*
        - *Giảm số lượng chỗ trống đi 1. Nếu không còn chỗ trống (buffer đầy), luồng bị chặn, đảm bảo không tràn buffer.*
    2. *pthread_mutex_lock(&gMutex): Khóa tài nguyên chia sẻ.*
        - *Đảm bảo không có luồng nào khác (Consumer) có thể truy cập mảng a và biến size trong khi Producer đang thao tác.*
    3. *Thêm phần tử (Critical Section):*
        - *a[size] = temp; và size++; (Thêm dữ liệu vào mảng và cập nhật biến đếm).*
    4. *pthread_mutex_unlock(&gMutex): Mở khóa tài nguyên.*
        - *Cho phép các luồng khác truy cập Critical Section.*
    5. *• sem_post(&sem_full_slots): Báo có hàng mới.*

- *Tăng số lượng phần tử đã lấp đầy lên 1, có thể đánh thức Consumer đang đợi.*

- *Hoạt động của ProcessB (Consumer - Tiêu thụ):*
  1. *sem_wait(&sem_full_slots): Đợi hàng có sẵn.*
     - *Giảm số lượng hàng có sẵn đi 1. Nếu không có hàng (buffer rỗng), luồng bị chặn, đảm bảo không lấy hàng không tồn tại.*
  2. *pthread_mutex_lock(&gMutex): Khóa tài nguyên chia sẻ.*
  3. *Lấy phần tử (Critical Section):*
     - *removedElement = a[size - 1]; và size--; (Lấy dữ liệu ra khỏi mảng và cập nhật biến đếm).*
  4. *pthread_mutex_unlock(&gMutex): Mở khóa tài nguyên.*
  5. *sem_post(&sem_empty_slots): Báo có chỗ trống mới.*
     - *Tăng số lượng chỗ trống lên 1, có thể đánh thức Producer đang đợi.*



Hình 6: Kết quả chạy bài 002b.

Nhận xét kết quả:

- *Kết quả chạy cho thấy quá trình Sản xuất và Tiêu thụ diễn ra đồng bộ và nhất quán.*
- *Tính toàn vẹn dữ liệu được đảm bảo:*

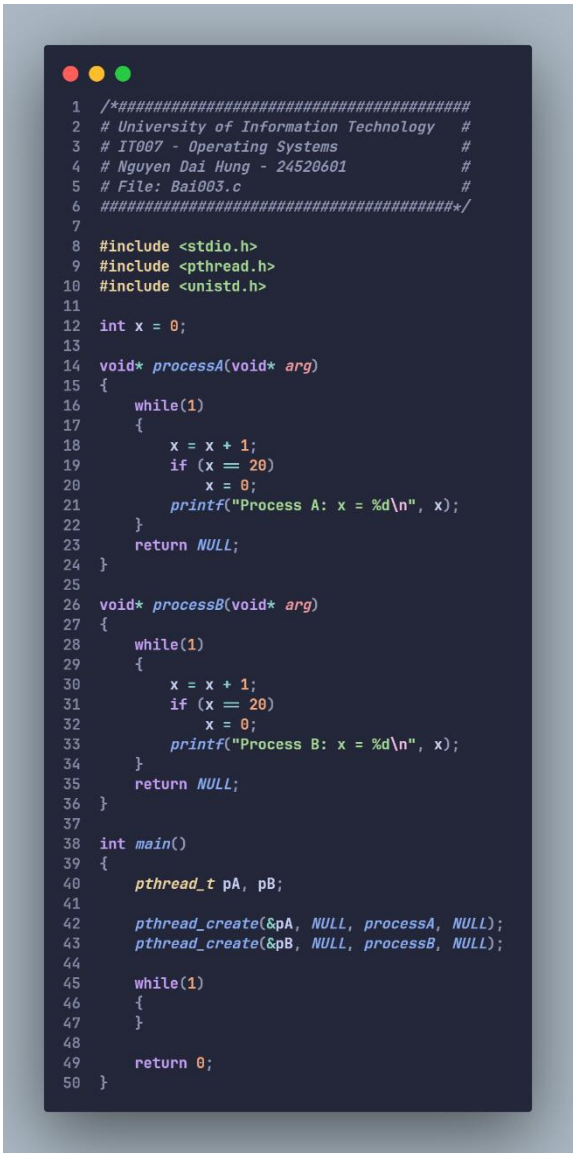
    - *Số lượng phần tử (Number of elements in array) luôn đúng (ví dụ: 10/10 khi Producer thêm, 9/10 khi Consumer lấy).*
- *Không xảy ra Race Condition: Không có hiện tượng mất cập nhật biến size hay dữ liệu bị ghi đè, chứng tỏ Mutex đã hoạt động hiệu quả để bảo vệ Critical Section.*
- *Như vậy, mã đã được đồng bộ hóa hoàn toàn chính xác về mặt cơ chế đảm bảo tính toàn vẹn và ràng buộc logic:*
    - *Không có lỗi Race Condition.*
    - *Không có lỗi vi phạm giới hạn.*

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.



```c
/*########################################
# University of Information Technology   #
# IT007 - Operating Systems              #
# Nguyen Dai Hung - 24520601             #
# File: Bai003.c                         #
########################################*/

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

int x = 0;

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

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

int main()
{
    pthread_t pA, pB;

    pthread_create(&pA, NULL, processA, NULL);
    pthread_create(&pB, NULL, processB, NULL);

    while(1)
    {
    }

    return 0;
}
```

Hình 7: Source code bài 003.

Giải thích code:

- *Bài toán yêu cầu cho 2 process A và B chạy song song với biến chung x = 0. Cả 2 process đều thực hiện:*
    - *Tăng x lên 1 đơn vị.*
    - *Reset x về 0 khi x= 20.*
    - *In giá trị x ra màn hình.*
- *Đầu tiên khai báo biến toàn cục x = 0 (đây là biến chung được cả 2 thread truy cập).*
- *Luồng hoạt động của cả ProcessA và Process B:*
    1. *Đọc giá trị hiện tại của x.*
    2. *Tính toán x = x + 1.*

3. *Ghi kết quả vào x.*
4. *Kiểm tra nếu x = 20 thì reset về 0.*
5. *In giá trị x ra màn hình.*
6. *Lặp lại vô hạn.*

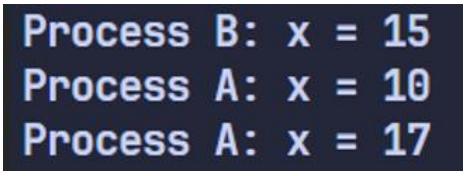
- *Đặc điểm của chương trình:*
  - *Cả 2 thread truy cập biến x mà không có cơ chế đồng bộ.*
  - *Thao tác x = x + 1 không phải atomic (gồm 3 bước: đoc → tính → ghi).*
  - *Có nguy cơ xảy ra race condition.*

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

Hình 8: Kết quả chạy bài 003.

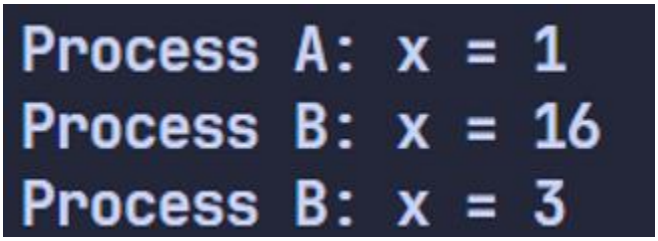
Nhận xét kết quả:

- *Hiện tượng số bị nhảy:*



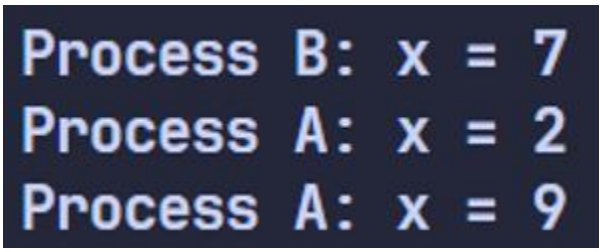
  - *Sau khi Process B in x = 15, giá trị x đáng lẽ phải tăng lên 16, nhưng Process A lại in x = 10. Điều này chứng tỏ một trong hai thread đã reset x về 0 nhưng thread kia không nhận biết được.*

- *Thứ tự giá trị không logic:*



  - *Giá trị x không tăng tuần tự như mong đợi (0→1→2→3→...→19→0). Có sự nhảy vọt và giảm đột ngột, chứng tỏ cả 2 thread đang đọc/ghi x ở các thời điểm khác nhau gây xung đột.*

- *Giá trị giảm bất thường:*



  - *Giá trị x có thể giảm đi mặc dù không có lệnh giảm trong code (chỉ có tăng và reset). Đây là dấu hiệu rõ ràng của race condition.*

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

```c
/*#######################################
# University of Information Technology   #
# IT007 - Operating Systems              #
# Nguyen Dai Hung - 24520601             #
# File: Bai004.c                         #
#######################################*/

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

int x = 0;
pthread_mutex_t mutex;

void* processA(void* arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);    // Khóa mutex trước khi truy cập x

        x = x + 1;
        if (x == 20)
            x = 0;
        printf("Process A: x = %d\n", x);

        pthread_mutex_unlock(&mutex);  // Mở khóa mutex

        usleep(100000);  // Delay 100ms
    }
    return NULL;
}

void* processB(void* arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);    // Khóa mutex trước khi truy cập x

        x = x + 1;
        if (x == 20)
            x = 0;
        printf("Process B: x = %d\n", x);

        pthread_mutex_unlock(&mutex);  // Mở khóa mutex

        usleep(100000);  // Delay 100ms
    }
    return NULL;
}

int main()
{
    pthread_t pA, pB;

    // Khởi tạo mutex
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&pA, NULL, processA, NULL);
    pthread_create(&pB, NULL, processB, NULL);

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

    // Hủy mutex
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Hình 9: Source code bài 004.

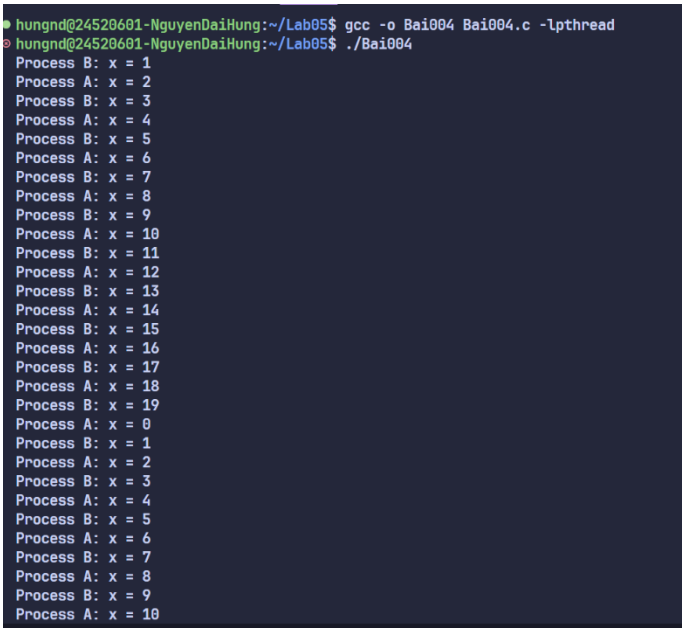
Giải thích code:

- *Yêu cầu bài toán đồng bộ với mutex để sửa lỗi bất hợp lý trong kết quả của mô hình Bài 3.*
- *Yêu cầu: Sử dụng mutex để đảm bảo chỉ có 1 thread được truy cập biến x tại một thời điểm, loại bỏ race condition.*
- *Đầu tiên ta cũng khai báo biến toàn cục x nhưng điểm khác biệt so với bài 3 là thêm biến mutex kiểu pthread_mutex_t để đồng bộ hóa.*
- *Hoạt động của Process A:*
  1. *pthread_mutex_lock(&mutex) - Yêu cầu khóa mutex:*
     - *Nếu mutex đang được Process B giữ → **Process A bị chặn**, đợi mutex được mở.*

- *Nếu mutex đang rảnh → Process A giữ mutex và tiếp tục.*

2. *Thực hiện các thao tác trong **critical section**:*
   - *Đọc giá trị x*
   - *Tính x + 1*
   - *Ghi kết quả vào x*
   - *Kiểm tra và reset nếu x = 20*
   - *In giá trị x*
3. *pthread_mutex_unlock(&mutex) - Mở khóa mutex, cho phép thread khác truy cập.*
4. *Lặp lại vô hạn*

- ***Hoạt động của Process B:** Tương tự Process A, đều phải xin phép mutex trước khi truy cập x.*
- *Các bước quan trọng:*
  - *pthread_mutex_init(&mutex, NULL) - Khởi tạo mutex với thuộc tính mặc định*
  - *Tạo 2 thread A và B chạy song song*
  - *Cả 2 thread đều được bảo vệ bởi cùng 1 mutex*
- *Dự đoán từ kịch bản:*
  - *Thread B KHÔNG THỂ vào critical section khi A đang giữ mutex.*
  - *Thread A KHÔNG THỂ vào critical section khi B đang giữ mutex*
  - *Giá trị x tăng tuần tự: 1 → 2 → 3 → 4 → 5 → 6 ...*



Hình 10: Kết quả chạy bài 004.

Nhận xét kết quả:

- *Giá trị x tăng tuần tự hoàn hảo.*
- *Cả 2 thread đều chạy.*
- *Không có race condition.*