

Name: Đỗ Hoàng Anh

ID: 22520041

Class: IT007.O212.1

OPERATING SYSTEM LAB 5 REPORT

SUMMARY

Task		Status	Page
Section 5.5	Ex 1	Hoàn thành	2
	Ex 2	Hoàn thành	4
	Ex 3	Hoàn thành	8
	Ex 4	Hoàn thành	11
Section 5.6	Ex 1	Hoàn thành	12

Self-scores: 10/10

Note: Export file to **PDF and name the file by following format:
LAB X – <Student ID>.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:
 $sells \leq products \leq sells + [4 \text{ số cuối của MSSV}]$.

```
C 5.5-1.c > processA(void *)
1 #include <stdio.h>
2 #include <semaphore.h>
3 #include <pthread.h>
4 #include <stdbool.h>
5
6 int sells = 0; // Biến đếm số lượng bán
7 int products = 0; // Biến đếm số lượng sản phẩm
8 sem_t sem, sem1; // Khai báo hai semaphore
9 pthread_mutex_t lock; // Khai báo mutex để khóa
10 bool stop = false; // Biến dừng quá trình
11
12 // Hàm xử lý cho processA
13 void *processA(void *mess)
14 {
15     while (1)
16     {
17         sem_wait(&sem); // Chờ sem được giải phóng
18         pthread_mutex_lock(&lock); // Khóa mutex
19         if (stop)
20         {
21             pthread_mutex_unlock(&lock); // Mở khóa mutex
22             sem_post(&sem1); // Giải phóng sem1
23             break; // Thoát vòng lặp
24         }
25
26         sells++; // Tăng biến đếm số lượng bán
27         if (sells == 41*2) // Kiểm tra nếu số lượng bán đạt 82
28         {
29             stop = true; // Dừng quá trình
30         }
31     }
32
33     printf("Sell = %d\n", sells); // In ra số lượng bán hiện tại
34     printf("Sell1 = %d\n", sells + 41); // In ra số lượng bán tiếp theo
35
36     sem_post(&sem1); // Giải phóng sem1
37 }
38 return NULL;
39 }
40
41 // Hàm xử lý cho processB
42 void *processB(void *mess)
43 {
44     while (1)
45     {
46         sem_wait(&sem1); // Chờ sem1 được giải phóng
47
48         pthread_mutex_lock(&lock); // Khóa mutex
49         if (stop)
50         {
51             pthread_mutex_unlock(&lock); // Mở khóa mutex
52             sem_post(&sem); // Giải phóng sem
53             break; // Thoát vòng lặp
54         }
55
56         if (products <= sells + 41) // Đảm bảo sản phẩm <= số lượng bán
57         {
58             products++; // Tăng biến đếm sản phẩm
59         }
60     }
61 }
62 return NULL;
63 }
```

```
Ubuntu 22.04.3 LTS
anhnh@anhAnh:~/hdh$ ./5.5-1
Products = 1
Products = 2
Products = 3
Products = 4
Products = 5
Products = 6
Products = 7
Products = 8
Products = 9
Products = 10
Products = 11
Products = 12
Products = 13
Products = 14
Products = 15
Products = 16
Products = 17
Products = 18
Products = 19
Products = 20
Products = 21
Products = 22
Products = 23
Products = 24
Products = 25
Products = 26
Products = 27
Products = 28
Products = 29
Products = 30
Products = 31

Products = 30
Products = 31
Products = 32
Products = 33
Products = 34
Products = 35
Products = 36
Products = 37
Products = 38
Products = 39
Products = 40
Products = 41
Sell = 1
Sell1 = 42
Sell = 2
Sell1 = 43
Sell = 3
Sell1 = 44
Products = 42
Products = 43
Products = 44
Sell = 4
Sell1 = 45
Sell = 5
Sell1 = 46
Sell = 6
Sell1 = 47
Products = 45
Products = 46
Products = 47
Sell = 7
Sell1 = 48
```

```

    {
        products++; // Tăng biến đếm sản phẩm
        pthread_mutex_unlock(&lock); // Mở khóa mutex

        printf("Products = %d\n", products); // In ra số lượng sản phẩm

        sem_post(&sem); // Giải phóng sem
    }
    else
    {
        pthread_mutex_unlock(&lock); // Mở khóa mutex
        sem_post(&sem); // Giải phóng sem
    }
}
return NULL;
}

int main()
{
    sem_init(&sem, 0, 0); // Khởi tạo semaphore sem
    sem_init(&sem1, 0, sells+41); // Khởi tạo semaphore sem1
    pthread_mutex_init(&lock, NULL); // Khởi tạo mutex

    pthread_t pA, pB;
    pthread_create(&pA, NULL, &processA, NULL); // Tạo thread cho processA
    pthread_create(&pB, NULL, &processB, NULL); // Tạo thread cho processB

    pthread_join(pA, NULL); // Chờ thread pA hoàn thành
    pthread_join(pB, NULL); // Chờ thread pB hoàn thành

    sem_destroy(&sem); // Hủy semaphore sem
    sem_destroy(&sem1); // Hủy semaphore sem1
    pthread_mutex_destroy(&lock); // Hủy mutex

    return 0;
}

bin/gcc -fdiagnostics-color=always -g /home/anhnh/hdh/5.5-1.c -o /home/anhnh/hdh/5.5-1
finished successfully.
Terminal will be reused by tasks, press any key to close it.
```

Sell1 = 49
Sell = 9
Sell1 = 50
Sell = 10
Sell1 = 51
Sell = 11
Sell1 = 52
Sell = 12
Sell1 = 53
Sell = 13
Sell1 = 54
Sell = 14
Sell1 = 55
Sell = 15
Sell1 = 56
Sell = 16
Sell1 = 57
Sell = 17
Sell1 = 58
Sell = 18
Sell1 = 59
Sell = 19
Sell1 = 60
Sell = 20
Sell1 = 61
Sell = 21
Sell1 = 62
Sell = 22
Sell1 = 63
Sell = 23
Sell1 = 64
Sell = 24

```

C 5.5-1.c > processA(void *)
42 void *processB(void *mess)
65 else
66 {
67     pthread_mutex_unlock(&lock); // Mở khóa mutex
68     sem_post(&sem); // Giải phóng sem
69 }
70 }
71 return NULL;
72 }
73
74 int main()
75 {
76     sem_init(&sem, 0, 0); // Khởi tạo semaphore sem
77     sem_init(&sem1, 0, sells+41); // Khởi tạo semaphore sem1
78     pthread_mutex_init(&lock, NULL); // Khởi tạo mutex
79
80     pthread_t pA, pB;
81     pthread_create(&pA, NULL, &processA, NULL); // Tạo thread cho processA
82     pthread_create(&pB, NULL, &processB, NULL); // Tạo thread cho processB
83
84     pthread_join(pA, NULL); // Chờ thread pA hoàn thành
85     pthread_join(pB, NULL); // Chờ thread pB hoàn thành
86
87     sem_destroy(&sem); // Hủy semaphore sem
88     sem_destroy(&sem1); // Hủy semaphore sem1
89     pthread_mutex_destroy(&lock); // Hủy mutex
90
91     return 0;
92 }
93

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

/usr/bin/gcc -fdiagnostics-color=always -g /home/anhnh/hdh/5.5-1.c -o /home/anhnh/hdh/5.5-1
Build finished successfully.
Terminal will be reused by tasks, press any key to close it.
```

Sell = 25
Sell1 = 66
Sell = 26
Sell1 = 67
Sell = 27
Sell1 = 68
Sell = 28
Sell1 = 69
Sell = 29
Sell1 = 70
Sell = 30
Sell1 = 71
Sell = 31
Sell1 = 72
Sell = 32
Sell1 = 73
Sell = 33
Sell1 = 74
Sell = 34
Sell1 = 75
Sell = 35
Sell1 = 76
Sell = 36
Sell1 = 77
Sell = 37
Sell1 = 78
Sell = 38
Sell1 = 79
Sell = 39
Sell1 = 80
Products = 48
Products = 49

Trong đó:

- Các biến toàn cục:
 - **sells** đếm số lượng bán.
 - **products** đếm số lượng sản phẩm.
 - **sem** và **sem1** là hai semaphore dùng để đồng bộ hóa giữa các tiến trình.
 - **lock** là một **mutex** dùng để bảo vệ các biến dùng chung.
 - **stop** là một biến **boolean** để dừng các tiến trình.
- Hàm **processA** thực hiện công việc tăng biến đếm **sells** và in ra giá trị của nó cùng với **sells + 41**. Khi **sells** đạt đến giá trị 82, nó đặt biến **stop** thành **true** để dừng vòng lặp.
- Hàm **processB** thực hiện công việc tăng biến đếm **products** và in ra giá trị của nó. Nó đảm bảo rằng **products** không vượt quá **sells + 41**. Nếu biến **stop** được đặt thành **true**, nó sẽ dừng vòng lặp.
- Hàm **main** khởi tạo các **semaphore** và **mutex**, tạo hai **thread processA** và **processB**, và chờ cho các **thread** này hoàn thành. Sau đó, nó hủy các **semaphore** và **mutex** đã khởi tạo.

Từ kết quả trên ta thấy biến **products** luôn luôn \geq **sells** và \leq **sell1**

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.

Khi chưa đồng bộ.

```
9 int a[MAX]; // Mảng a toàn cục
10 int n = 0; // Số lượng phần tử hiện tại trong mảng
11 sem_t sem_producer, sem_consumer; // Semaphore để đồng bộ hóa
12 pthread_mutex_t lock; // Mutex để bảo vệ biến dùng chung
13
14 void *producer(void *arg) {
15     while (1) {
16         int num = rand() % 100; // Sinh số ngẫu nhiên
17
18         // sem_wait(&sem_producer); // Chờ semaphore của producer
19
20         // pthread_mutex_lock(&lock); // Khóa mutex
21
22         a[n] = num; // Thêm phần tử vào mảng
23         n++; // Tăng số lượng phần tử
24         printf("Added %d to array. Number of elements: %d\n", num, n);
25         for(int i = 0; i < n; i++)
26             printf(" %d ", a[i]);
27         printf("\n");
28
29         // pthread_mutex_unlock(&lock); // Mở khóa mutex
30
31         // sem_post(&sem_consumer); // Giải phóng semaphore của consumer
32         // sleep(1); // Chờ 1 giây trước khi sinh số tiếp theo
33     }
34     return NULL;
35 }
36
37 void *consumer(void *arg) {
38     while (1) {
39         // sem_wait(&sem_consumer); // Chờ semaphore của consumer
40
41         // pthread_mutex_lock(&lock); // Khóa mutex
42         if(n <= 0) {
43             printf("Nothing in array a\n");
44         }
45
46         int value = a[n-1];
47         n--; // Giảm số lượng phần tử
48         printf("Removed %d from array. Number of elements: %d\n", value, n);
49         for(int i = 0; i < n; i++)
50             printf(" %d ", a[i]);
51         printf("\n");
52
53         // pthread_mutex_unlock(&lock); // Mở khóa mutex
54
55         // sem_post(&sem_producer); // Giải phóng semaphore của producer
56         // sleep(2); // Chờ 2 giây trước khi lấy phần tử tiếp theo
57     }
58     return NULL;
59 }
60
61 int main() {
62     srand(time(NULL)); // Khởi tạo seed cho hàm rand()
63
64     sem_init(&sem_producer, 0, MAX); // Khởi tạo semaphore của producer
65     sem_init(&sem_consumer, 0, 0); // Khởi tạo semaphore của consumer
66     pthread_mutex_init(&lock, NULL); // Khởi tạo mutex
67
68     pthread_t producer_thread, consumer_thread;
69     pthread_create(&producer_thread, NULL, producer, NULL);
70     pthread_create(&consumer_thread, NULL, consumer, NULL);
71     pthread_join(producer_thread, NULL);
72     pthread_join(consumer_thread, NULL);
73
74     return 0;
75 }
```

```
anhnh@anhAnh:~/hdh$ ./5.5-2
Nothing in array a
Removed 1 from array. Number of elements: 0

Nothing in array a
Removed 0 from array. Number of elements: -1

Nothing in array a
Removed 0 from array. Number of elements: -2

Nothing in array a
Removed 0 from array. Number of elements: -3

Nothing in array a
Removed 0 from array. Number of elements: -4

Nothing in array a
Removed 0 from array. Number of elements: -5

Nothing in array a
Removed 0 from array. Number of elements: -6

Nothing in array a
Removed 0 from array. Number of elements: -7

Nothing in array a
Removed 0 from array. Number of elements: -8

Nothing in array a
Removed 0 from array. Number of elements: -9

Nothing in array a

Nothing in array a
Removed 0 from array. Number of elements: -10

Nothing in array a
Removed 0 from array. Number of elements: -11

Nothing in array a
Removed 0 from array. Number of elements: -12

Nothing in array a
Removed 21847 from array. Number of elements: -13

Nothing in array a
Removed -793440248 from array. Number of elements: -14

Nothing in array a
Removed 0 from array. Number of elements: -15

Nothing in array a
Removed 0 from array. Number of elements: -16

Nothing in array a
Removed 32659 from array. Number of elements: -17

Nothing in array a
Removed 1825876384 from array. Number of elements: -18

Nothing in array a
Removed 0 from array. Number of elements: -19

Nothing in array a
```


- Sau khi chạy đoạn code chưa đồng bộ, chúng ta có thể nhận thấy một số lỗi chính sau:
 - Lỗi vùng tranh chấp:
 - Khi tiến trình A thực hiện các thao tác trên vùng tranh chấp, tiến trình B cũng thực hiện các thao tác của mình trên cùng vùng tranh chấp. Điều này dẫn đến các điều kiện đua (race conditions) và gây ra lỗi.
 - Lỗi về giới hạn phân tử:
 - Ban đầu, chúng ta nhập n là số phần tử tối đa trong mảng. Tuy nhiên, do không có ràng buộc điều kiện bằng semaphore nên số phần tử có thể vượt quá giới hạn mà chúng ta nhập vào. Điều này có thể dẫn đến việc ghi đè lên bộ nhớ ngoài phạm vi của mảng.
 - Lỗi khi loại bỏ phần tử trong mảng:
 - Khi không còn phần tử nào trong mảng, tiến trình vẫn tiếp tục thực thi, dẫn đến việc đếm số phần tử sẽ ra số âm. Điều này không đúng và có thể gây ra lỗi nghiêm trọng trong chương trình.

Đã được đồng bộ:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #include <time.h>
6
7 #define MAX 100 // Giới hạn kích thước mảng
8
9 int a[MAX]; // Mảng a toàn cục
10 int n = 0; // Số lượng phần tử hiện tại trong mảng
11 sem_t sem_producer, sem_consumer; // Semaphore để đồng bộ hóa
12 pthread_mutex_t lock; // Mutex để bảo vệ biến dùng chung
13
14 void *producer(void *arg) {
15     while (1) {
16         int num = rand() % 100; // Sinh số ngẫu nhiên
17
18         sem_wait(&sem_producer); // Chờ semaphore của producer
19
20         pthread_mutex_lock(&lock); // Khóa mutex
21
22         a[n] = num; // Thêm phần tử vào mảng
23         n++; // Tăng số lượng phần tử
24         printf("Added %d to array. Number of elements: %d\n", num, n);
25         for(int i = 0; i < n; i++) {
26             printf("%d ", a[i]);
27         }
28         printf("\n");
29     }
30 }
31
32 void *consumer(void *arg) {
33     while (1) {
34         sem_wait(&sem_consumer); // Chờ semaphore của consumer
35
36         pthread_mutex_lock(&lock); // Khóa mutex
37
38         if (n < 1) {
39             printf("No elements in array. Producer must add elements first.\n");
40             sem_post(&sem_producer); // Giải phóng semaphore của producer
41             continue;
42         }
43
44         int val = a[0]; // Lấy phần tử đầu tiên
45         printf("Removed %d from array. Number of elements: %d\n", val, n);
46         for(int i = 1; i < n; i++) {
47             a[i-1] = a[i]; // Dịch mảng về phía trước
48         }
49         n--; // Giảm số lượng phần tử
50         printf("\n");
51         sem_post(&sem_consumer); // Giải phóng semaphore của consumer
52     }
53 }
54
55 int main() {
56     pthread_t producer_thread, consumer_thread;
57     sem_init(&sem_producer, 0, 1);
58     sem_init(&sem_consumer, 0, 1);
59     pthread_mutex_init(&lock, NULL);
60     producer_thread = pthread_create(&producer_thread, NULL, producer, NULL);
61     consumer_thread = pthread_create(&consumer_thread, NULL, consumer, NULL);
62     pthread_join(producer_thread, NULL);
63     pthread_join(consumer_thread, NULL);
64     printf("Program finished successfully.\n");
65     return 0;
66 }

```

Terminal output:

```

Added 24 to array. Number of elements: 99
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24
Added 38 to array. Number of elements: 100
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24 38
Removed 38 from array. Number of elements: 99
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24
Added 83 to array. Number of elements: 100
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24 83
Removed 83 from array. Number of elements: 99
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43

```

```
14 void *producer(void *arg) {
31     pthread_mutex_unlock(&lock); // Mở khóa mutex
32     sem_post(&sem_consumer); // Giải phóng semaphore của consumer
33
34 }
35 return NULL;
36 }
37
38 void *consumer(void *arg) {
39     while (1) {
40         sem_wait(&sem_consumer); // Chờ semaphore của consumer
41
42         pthread_mutex_lock(&lock); // Khóa mutex
43         if(n <= 0) {
44             printf("Nothing in array a\n");
45         }
46
47         int value = a[n-1];
48         n--; // Giảm số lượng phần tử
49         printf("Removed %d from array. Number of elements: %d\n", value, n);
50         for(int i = 0; i < n; i++) {
51             printf(" %d ", a[i]);
52         }
53         pthread_mutex_unlock(&lock); // Mở khóa mutex
54         sem_post(&sem_producer);
55
56         printf("\n");
57     }
58     return NULL;
59 }

```

```

33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24
Added 68 to array. Number of elements: 100
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24 68
Removed 68 from array. Number of elements: 99
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24
Removed 24 from array. Number of elements: 98
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24
Added 21 to array. Number of elements: 99
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44

```

```

18 void *consumer(void *arg) {
22     }
23     pthread_mutex_unlock(&lock); // Mở khóa mutex
24     sem_post(&sem_producer);
25
26     printf("\n");
27 }
28 return NULL;
29 }
30
31 int main() {
32     srand(time(NULL)); // Khởi tạo seed cho hàm rand()
33
34     sem_init(&sem_producer, 0, MAX); // Khởi tạo semaphore của producer
35     sem_init(&sem_consumer, 0, 0); // Khởi tạo semaphore của consumer
36     pthread_mutex_init(&lock, NULL); // Khởi tạo mutex
37
38     pthread_t tid1, tid2;
39     pthread_create(&tid1, NULL, producer, NULL); // Tạo thread producer
40     pthread_create(&tid2, NULL, consumer, NULL); // Tạo thread consumer
41
42     pthread_join(tid1, NULL); // Chờ thread producer hoàn thành
43     pthread_join(tid2, NULL); // Chờ thread consumer hoàn thành
44
45     sem_destroy(&sem_producer); // Hủy semaphore của producer
46     sem_destroy(&sem_consumer); // Hủy semaphore của consumer
47     pthread_mutex_destroy(&lock); // Hủy mutex
48
49     return 0;
50 }

```

```

33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24
Added 68 to array. Number of elements: 100
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24 68
Removed 68 from array. Number of elements: 99
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24
Removed 24 from array. Number of elements: 98
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44
81 98 53 27 48 92 48 59 47 53 84 55 24
Added 21 to array. Number of elements: 99
60 27 65 24 67 85 77 71 54 41 20 58 94 67 47 89
33 83 46 53 81 79 50 74 57 21 96 70 69 74 82 48
6 16 29 83 87 84 24 7 42 71 26 41 60 67 74 43
7 76 45 78 50 2 51 98 24 72 25 6 21 17 12 89 98
76 82 72 36 76 95 14 18 7 81 92 50 47 72 79 44

```

- Việc đồng bộ hóa được thực hiện thông qua việc sử dụng semaphore và mute:

- **Semaphore của Producer: `sem_wait(&sem_producer)`**; được gọi trước khi Producer thêm một phần tử vào mảng. Điều này đảm bảo rằng Producer sẽ chờ cho đến khi có vị trí trống trong mảng để thêm phần tử.
- **`sem_post(&sem_consumer)`**; được gọi sau khi Producer thêm một phần tử vào mảng. Điều này thông báo cho Consumer rằng có một phần tử mới được thêm vào mảng và nó có thể lấy phần tử từ mảng.
- **Semaphore của Consumer: `sem_wait(&sem_consumer)`**; được gọi trước khi Consumer thực hiện việc loại bỏ một phần tử khỏi mảng. Điều này đảm bảo rằng Consumer sẽ chờ cho đến khi có ít nhất một phần tử trong mảng để loại bỏ.
- **`sem_post(&sem_producer)`**; được gọi sau khi Consumer loại bỏ một phần tử khỏi mảng. Điều này thông báo cho Producer rằng có một vị trí trống trong mảng và nó có thể thêm phần tử mới vào.
- **Mutex: `pthread_mutex_lock(&lock)`**; được gọi trước khi Producer hoặc Consumer truy cập hoặc sửa đổi mảng. Điều này đảm bảo rằng chỉ một tiến trình được phép thực hiện thao tác trên mảng tại một thời điểm, tránh tranh chấp và hậu quả không mong muốn.
- **`pthread_mutex_unlock(&lock)`**; được gọi sau khi Producer hoặc Consumer hoàn thành thao tác trên mảng. Điều này mở khóa mutex và cho phép các tiến trình khác tiếp tục thực hiện thao tác trên mảng.

Qua việc sử dụng semaphore và mutex như trên, chương trình đảm bảo tính nhất quán và đồng bộ hóa giữa Producer và Consumer khi truy cập và thay đổi mảng a.

3. Cho 2 process A và B chạy song song như sau:
`int x = 0;`

PROCESS A	PROCESS B
<pre>processA() { while(1){ x = x + 1; if (x == 20) x = 0; print(x); } }</pre>	<pre>processB() { while(1){ x = x + 1; if (x == 20) x = 0; print(x); } }</pre>

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

Khi chạy chương trình:

```

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

int x = 0; // Biến x toàn cục được sử dụng bởi cả hai tiến trình

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

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

int main() {
    pthread_t tidA, tidB;

    pthread_create(&tidA, NULL, processA, NULL);
    pthread_create(&tidB, NULL, processB, NULL);

    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);

    return 0;
}

```

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

```
C 5.5-1.c C 5.5-2.c X C 5.5-3.c X
C 5.5-3.c > processB(void *)
7 void *processA(void *arg) {
15 }
16
17 // Hàm thực thi cho tiến trình B
18 void *processB(void *arg) {
19     while (1) {
20         x = x + 1;
21         if (x == 20)
22             x = 0;
23         printf("Process B: %d\n", x);
24     }
25     return NULL;
26 }
27
28 int main() {
29     pthread_t tidA, tidB;
30
31     // Tạo tiến trình A
32     pthread_create(&tidA, NULL, processA, NULL);
33
34     // Tạo tiến trình B
35     pthread_create(&tidB, NULL, processB, NULL);
36
37     // Chờ tiến trình A và tiến trình B hoàn thành
38     pthread_join(tidA, NULL);
39     pthread_join(tidB, NULL);
40
41     return 0;
42 }
43

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
/usr/bin/gcc -fdiagnostics-color=always -g /home/anhnh/hdh/5.5-3.c -o /home/anhnh/hdh/
Build finished successfully.
Terminal will be reused by tasks, press any key to close it.

Ubuntu 22.04.3 LTS
Process B: 19
Process B: 0
Process B: 1
Process B: 2
Process B: 3
Process B: 4
Process B: 5
Process B: 6
Process B: 7
Process B: 8
Process B: 9
Process A: 17
Process A: 11
Process A: 12
Process A: 13
Process A: 14
Process A: 15
Process A: 16
Process A: 17
Process A: 18
Process A: 19
Process A: 0
Process A: 1
Process A: 2
Process A: 3
Process A: 4
Process A: 5
Process A: 6
Process A: 7
Process A: 8
Process A: 9
Process A: 10
```

Khi chạy chương trình trên, chúng ta đã nhận thấy một số vấn đề, trong đó, lỗi về vùng tranh chấp là một trong những vấn đề quan trọng nhất cần được phân tích. Khi tiến trình A và B thực hiện cùng một loạt lệnh, sự xung đột xảy ra khi cả hai tiến trình cố gắng thực hiện vùng tranh chấp của mình đồng thời. Điều này dẫn đến kết quả không nhất quán và không đoán trước được.

Ví dụ, sau khi tiến trình B hoàn thành việc thực thi $x = 15$, chúng ta có thể mong đợi rằng tiến trình A sẽ tiếp tục với giá trị $x = 16$, nhưng thực tế lại là $x = 5$. Tương tự, sau khi tiến trình B thực hiện $x = 9$, chúng ta có thể mong đợi rằng tiến trình A sẽ tiếp tục với $x = 10$, nhưng thực tế lại là $x = 17$.

Điều này là do cả hai tiến trình thực hiện các lệnh một cách không đồng bộ và không được đồng bộ hóa đúng cách, dẫn đến việc xảy ra lỗi vùng tranh chấp. Để khắc phục vấn đề này, chúng ta cần phải sử dụng các phương pháp đồng bộ hóa như semaphore hoặc mutex để đảm bảo rằng chỉ có một tiến trình được phép thực thi vùng tranh chấp một cách an toàn tại mỗi thời điểm, đồng thời đảm bảo tính nhất quán của dữ liệu và kết quả.

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

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

int x = 0; // Biến x toàn cục được sử dụng bởi cả hai tiến trình
pthread_mutex_t mutex; // Mutex để đồng bộ hóa
// Hàm thực thi cho tiến trình A
void *processA(int *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("Process A: %d\n", x);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

// Hàm thực thi cho tiến trình B
void *processB(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        x = x + 1;
        if (x == 20)
            x = 0;
        printf("Process B: %d\n", x);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

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

```
5.5-3.c > processA(void *)
20 void *processB(void *arg) {
26     printf("Process B: %d\n", x);
27     pthread_mutex_unlock(&mutex);
28 }
29 return NULL;
30 }
31
32 int main() {
33     pthread_t tidA, tidB;
34
35     pthread_mutex_init(&mutex, NULL); // Khởi tạo mutex
36
37     // Tạo tiến trình A
38     pthread_create(&tidA, NULL, processA, NULL);
39
40     // Tạo tiến trình B
41     pthread_create(&tidB, NULL, processB, NULL);
42
43     // Chờ tiến trình A và tiến trình B hoàn thành
44     pthread_join(tidA, NULL);
45     pthread_join(tidB, NULL);
46
47     pthread_mutex_destroy(&mutex); // Hủy mutex
48
49     return 0;
50 }
51
```

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

- **Biến Mutex (`pthread_mutex_t mutex`):** Mutex được sử dụng để đảm bảo rằng chỉ có một tiến trình được phép thay đổi giá trị của biến `x` tại một thời điểm. Khi một tiến trình đã khóa mutex, tiến trình khác phải đợi cho đến khi mutex được mở khóa trước khi họ có thể truy cập biến `x`.
- **Khóa Mutex (`pthread_mutex_lock(&mutex)`):** Trước khi thực hiện bất kỳ thay đổi nào vào biến `x`, mỗi tiến trình phải khóa mutex bằng lệnh `pthread_mutex_lock(&mutex)`. Điều này đảm bảo rằng chỉ có một tiến trình được phép thay đổi giá trị của biến `x` tại một thời điểm.
- **Mở Khóa Mutex (`pthread_mutex_unlock(&mutex)`):** Sau khi thực hiện xong các thay đổi vào biến `x`, tiến trình phải mở khóa mutex bằng lệnh `pthread_mutex_unlock(&mutex)`. Điều này cho phép các tiến trình khác có thể khóa mutex và truy cập biến `x`.

Từ kết quả trên ta đã thấy biến `x` có sự liên tiếp giữa 2 processA và processB

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) \rightarrow (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

Kết quả khi chạy:

[illegible]

- **sem_ab, sem_cd, sem_ef, sem_g:** Là các semaphore được sử dụng để đồng bộ hóa các tiến trình.

- **Hàm processAB:**
 - Thực hiện các phép tính $w = x1 * x2$ và $v = x3 * x4$.
 - Sau khi thực hiện xong, gửi semaphore `sem_cd` để báo cho tiến trình `processCD` biết rằng các giá trị w và v đã được tính toán.
- **Hàm processCD:**
 - Thực hiện các phép tính $y = v * x5$ và $z = v * x6$.
 - Sau khi thực hiện xong, gửi semaphore `sem_ef` để báo cho tiến trình `processEF` biết rằng các giá trị y và z đã được tính toán.
- **Hàm processEF:**
 - Thực hiện các phép tính $y = w * y$ và $z = w * z$.
 - Sau khi thực hiện xong, gửi semaphore `sem_g` để báo cho tiến trình `processG` biết rằng các giá trị y và z đã được tính toán.
- **Hàm processG:**
 - Thực hiện phép tính $ans = y + z$.
 - Sau khi thực hiện xong, gửi semaphore `sem_ab` để báo cho tiến trình `processAB` biết rằng kết quả ans đã được tính toán và có thể bắt đầu một chu trình tính toán mới.
- **Hàm main:**
 - Khởi tạo các semaphore bằng `sem_init()` với giá trị ban đầu là 0.
 - Tạo và chờ các tiến trình con (`processAB`, `processCD`, `processEF`, `processG`) bằng `pthread_create()` và `pthread_join()`.
 - Hủy các semaphore bằng `sem_destroy()` sau khi sử dụng xong.