

## BÀI BÁO CÁO THỰC HÀNH 5 – HỆ ĐIỀU HÀNH

CLASS : IT007.O26.1

TEACHER : MR. ĐOÀN DUY

Date: 10/06/2024

Reported by: Trần Ngọc Ánh | 22520077 | [22520077@gm.uit.edu.vn](mailto:22520077@gm.uit.edu.vn)

### SUMMARY

Task		Status	Page
Thực hành	Bài 1	Hoàn thành	1
	Bài 2	Hoàn thành	4
	Bài 3	Hoàn thành	9
	Bài 4	Hoàn thành	11
Ôn tập	Bài 1	Hoàn thành	13

#### A. Bài tập thực hành

- Hiện thực hóa mô hình trong ví dụ 5.3.1.2, tuy nhiên thay bằng điều kiện sau:  $sells \leq products \leq sells + [4 \text{ số cuối của MSSV}]$

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

int sells = 0, products = 0;
sem_t sem, sem1;

void* processA(void* mes) {
    while (1) {
        sem_wait(&sem);
        if (products >= 100) {
            sem_post(&sem1); // Đảm bảo tiến trình B có thể thoát nếu chờ
            break;
        }
        sells++;
        printf("SELL = %d\n", sells);
    }
}
```

```

        printf("SELL_1 = %d\n", sells + 77);
        sem_post(&sem1);
    }
    return NULL;
}

void* processB(void* mess) {
    while (1) {
        sem_wait(&sem1);
        if (products >= 100) {
            sem_post(&sem); // Đảm bảo tiến trình A có thể thoát nếu chờ
            break;
        }
        products++;
        printf("PRODUCT = %d\n", products);
        sem_post(&sem);
    }
    return NULL;
}

int main() {
    sem_init(&sem, 0, 0);
    // Khởi tạo thành 1 cho phép tiến trình B bắt đầu trước
    sem_init(&sem1, 0, 1);
    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);
    sem_destroy(&sem1);

    return 0;
}

```

- 2 semaphore để kiểm tra điều kiện gồm sem và sem1, 2 hàm process để thực hiện công việc sản xuất hàng và bán hàng.
- Ta sẽ gán cho 2 biến sells và products đều bằng 0, khai báo 2 semaphore với sem = 0 và sem1 = 77 (4 số cuối mã số sinh viên 22520077).

- processA: bán hàng (SELL)

+ **sem\_wait(&sem)** để kiểm tra giá trị của sem, nếu sem = 0 thì block, nếu sem > 0 thì thực hiện lệnh sem = sem - 1. Sau đó thực hiện sells++ (muốn bán được hàng thì phải có hàng trước). Vậy hàm sem\_wait() mục đích là để kiểm tra xem đã có hàng hay chưa.

+ **sem\_post(&sem1)** tăng giá trị của sem1 lên làm điều kiện cho hàm processB.

- processB: sản xuất hàng (PRODUCT)

+ **sem\_wait(&sem1)** kiểm tra sem1 = 0 (tương tự process A) hay không, mục đích kiểm tra products < sells + 77 hay không để thỏa mãn điều kiện.

+ Tăng biến products lên và sử dụng hàm **sem\_post(&sem)** để thông báo cho processA biết là đã có hàng.

```
PRODUCT = 1
SELL = 1
SELL_1 = 78
PRODUCT = 2
SELL = 2
SELL_1 = 79
PRODUCT = 3
SELL = 3
SELL_1 = 80
PRODUCT = 4
SELL = 4
SELL_1 = 81
PRODUCT = 5
SELL = 5
SELL_1 = 82
PRODUCT = 6
SELL = 6
SELL_1 = 83
PRODUCT = 7
SELL = 7
SELL_1 = 84
PRODUCT = 8
SELL = 8
SELL_1 = 85
PRODUCT = 9
SELL = 9
SELL_1 = 86
PRODUCT = 10
SELL = 10
SELL_1 = 87
```

```
PRODUCT = 11
SELL = 11
SELL_1 = 88
PRODUCT = 12
SELL = 12
SELL_1 = 89
PRODUCT = 13
SELL = 13
SELL_1 = 90
PRODUCT = 14
SELL = 14
SELL_1 = 91
PRODUCT = 15
SELL = 15
SELL_1 = 92
PRODUCT = 16
SELL = 16
SELL_1 = 93
PRODUCT = 17
SELL = 17
SELL_1 = 94
PRODUCT = 18
SELL = 18
SELL_1 = 95
PRODUCT = 19
SELL = 19
SELL_1 = 96
PRODUCT = 20
SELL = 20
SELL_1 = 97
```

```
PRODUCT = 21
SELL = 21
SELL_1 = 98
PRODUCT = 22
SELL = 22
SELL_1 = 99
PRODUCT = 23
SELL = 23
SELL_1 = 100
PRODUCT = 24
SELL = 24
SELL_1 = 101
PRODUCT = 25
SELL = 25
SELL_1 = 102
PRODUCT = 26
SELL = 26
SELL_1 = 103
PRODUCT = 27
SELL = 27
SELL_1 = 104
PRODUCT = 28
SELL = 28
SELL_1 = 105
PRODUCT = 29
SELL = 29
SELL_1 = 106
PRODUCT = 30
SELL = 30
SELL_1 = 107
```

```
PRODUCT = 31
SELL = 31
SELL_1 = 108
PRODUCT = 32
SELL = 32
SELL_1 = 109
PRODUCT = 33
SELL = 33
SELL_1 = 110
PRODUCT = 34
SELL = 34
SELL_1 = 111
PRODUCT = 35
SELL = 35
SELL_1 = 112
PRODUCT = 36
SELL = 36
SELL_1 = 113
PRODUCT = 37
SELL = 37
SELL_1 = 114
PRODUCT = 38
SELL = 38
SELL_1 = 115
PRODUCT = 39
SELL = 39
SELL_1 = 116
PRODUCT = 40
SELL = 40
SELL_1 = 117
```

```
PRODUCT = 41
SELL = 41
SELL_1 = 118
PRODUCT = 42
SELL = 42
SELL_1 = 119
PRODUCT = 43
SELL = 43
SELL_1 = 120
PRODUCT = 44
SELL = 44
SELL_1 = 121
PRODUCT = 45
SELL = 45
SELL_1 = 122
PRODUCT = 46
SELL = 46
SELL_1 = 123
PRODUCT = 47
SELL = 47
SELL_1 = 124
PRODUCT = 48
SELL = 48
SELL_1 = 125
PRODUCT = 49
SELL = 49
SELL_1 = 126
PRODUCT = 50
SELL = 50
SELL_1 = 127
```

Nhìn vào kết quả chạy của 100 products, ta thấy products luôn luôn lớn hơn hoặc bằng sells và luôn luôn nhỏ hơn sells + 77 (sell\_1), thỏa mãn được điều kiện mà bài tập đã nêu ra.

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:
  - a. 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.
  - b. 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”.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h> // For sleep function

int* a;
int n;
int iNum = 0;
pthread_mutex_t mutex;

void Arrange(int* a, int x) {
    for (int i = x; i < iNum - 1; ++i) {
        a[i] = a[i + 1];
    }
    iNum--;
}

void* processA(void* mess) {
    while (1) {
        pthread_mutex_lock(&mutex);
        if (iNum < n) {
            int new_value = rand() % 100; // Tạo số ngẫu nhiên từ 0
            // đến 99
        }
    }
}
```

```

        a[iNum] = new_value;
        iNum++;
        printf("Thêm phần tử: %d\n", new_value);
        printf("Số lượng phần tử sau khi thêm: %d\n", iNum);
    }
    sleep(1); // Nghỉ 1 giây để tránh thêm phần tử quá nhanh
    pthread_mutex_unlock(&mutex);
}
return NULL;
}

void* processB(void* mess) {
    while (1) {
        pthread_mutex_lock(&mutex);
        if (iNum == 0) {
            printf("Nothing in array a\n");
        } else {
            int r = rand() % iNum;
            int removed_value = a[r];
            Arrange(a, r);
            printf("Loại bỏ phần tử: %d\n", removed_value);
            printf("Số lượng phần tử sau khi bỏ: %d\n", iNum);
        }
        sleep(1); // Nghỉ 1 giây để tránh lấy phần tử quá nhanh
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main() {
    printf("Nhập số lượng phần tử: ");
    scanf("%d", &n);

    a = (int*)malloc(n * sizeof(int));
    pthread_mutex_init(&mutex, NULL);

    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);
    free(a);
    return 0;
}

```

- Race Condition: 2 thread processA và processB có thể cùng thao tác vào mảng a mà không có sự đồng bộ hóa. Nếu processA thực hiện việc tăng iNum và processB cùng lúc thực hiện giảm iNum, có thể xảy ra tình huống không lường trước được và dẫn đến kết quả không chính xác hoặc lỗi.
- Thiếu bảo vệ đối với biến iNum: iNum được truy cập và thay đổi bởi cả hai thread mà không có bất kỳ cơ chế bảo vệ nào. Điều này có thể dẫn đến xung đột giữa các thao tác đọc và ghi trên iNum.

```
ngocanh-22520077@TNANH-D53R8MB9:~$ gcc lab5_2a.c -o 2a
ngocanh-22520077@TNANH-D53R8MB9:~$ ./2a
Nhập số lượng phần tử: 7
Thêm phần tử: 83
Số lượng phần tử sau khi thêm: 1
Thêm phần tử: 86
Số lượng phần tử sau khi thêm: 2
Thêm phần tử: 77
Số lượng phần tử sau khi thêm: 3
Thêm phần tử: 15
Số lượng phần tử sau khi thêm: 4
Thêm phần tử: 93
Số lượng phần tử sau khi thêm: 5
Thêm phần tử: 35
Số lượng phần tử sau khi thêm: 6
Thêm phần tử: 86
Số lượng phần tử sau khi thêm: 7
Loại bỏ phần tử: 77
Số lượng phần tử sau khi bỏ: 6
Loại bỏ phần tử: 93
Số lượng phần tử sau khi bỏ: 5
Loại bỏ phần tử: 86
Số lượng phần tử sau khi bỏ: 4
Loại bỏ phần tử: 35
Số lượng phần tử sau khi bỏ: 3
Loại bỏ phần tử: 15
Số lượng phần tử sau khi bỏ: 2
Loại bỏ phần tử: 83
Số lượng phần tử sau khi bỏ: 1
Loại bỏ phần tử: 86
Số lượng phần tử sau khi bỏ: 0
Nothing in array a
```

Sau khi đã được đồng bộ:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <semaphore.h>
#include <unistd.h> // For sleep function

int* a;
int n;
int iNum = 0;
pthread_mutex_t mutex;
sem_t sem1, sem2;

void* processA(void* mess) {
    while (1) {
        sem_wait(&sem2);
        pthread_mutex_lock(&mutex);
        srand(time(NULL));
        // Tạo số ngẫu nhiên từ 0 đến 99
        int new_value = rand() % 100;
        a[iNum] = new_value;
        iNum++;
        printf("Thêm phần tử: %d\n", new_value);
        printf("Số lượng phần tử sau khi thêm: %d\n", iNum);
        sleep(1); // Nghỉ 1 giây để tránh thêm phần tử quá nhanh
        sem_post(&sem1);
        pthread_mutex_unlock(&mutex);
    }
}

void* processB(void* mess) {
    while (1) {
        sem_wait(&sem1);
        pthread_mutex_lock(&mutex);
        int removed_value = a[iNum - 1];
        a[iNum - 1] = 0;
        iNum--;
        printf("Loại bỏ phần tử: %d\n", removed_value);
        printf("Số lượng phần tử sau khi bỏ: %d\n", iNum);
        if (iNum == 0) {
            printf("Nothing in array a\n");
        }
        sleep(1); // Nghỉ 1 giây để tránh lấy phần tử quá nhanh
        sem_post(&sem2);
    }
}
```



```

        pthread_mutex_unlock(&mutex);
    }
}

int main() {
    printf("Nhập số lượng phần tử: ");
    scanf("%d", &n);
    a = (int*)malloc(n * sizeof(int));
    sem_init(&sem1, 0, 0);
    sem_init(&sem2, 0, n);
    pthread_mutex_init(&mutex, NULL);
    pthread_t pA, pB;
    pthread_create(&pA, NULL, &processA, NULL);
    pthread_create(&pB, NULL, &processB, NULL);
    while (1) {};
    return 0;
}

```

```

ngocanh-22520077@TNANH-D53R8MB9:~$ gcc lab5_2b.c -o 2b
ngocanh-22520077@TNANH-D53R8MB9:~$ ./2b

```

```

Nhập số lượng phần tử: 7
Thêm phần tử: 15
Số lượng phần tử sau khi thêm: 1
Thêm phần tử: 14
Số lượng phần tử sau khi thêm: 2
Thêm phần tử: 40
Số lượng phần tử sau khi thêm: 3
Thêm phần tử: 4
Số lượng phần tử sau khi thêm: 4
Thêm phần tử: 40
Số lượng phần tử sau khi thêm: 5
Thêm phần tử: 33
Số lượng phần tử sau khi thêm: 6
Thêm phần tử: 74
Số lượng phần tử sau khi thêm: 7
Loại bỏ phần tử: 74
Số lượng phần tử sau khi bỏ: 6
Loại bỏ phần tử: 33
Số lượng phần tử sau khi bỏ: 5
Loại bỏ phần tử: 40
Số lượng phần tử sau khi bỏ: 4
Loại bỏ phần tử: 4
Số lượng phần tử sau khi bỏ: 3
Loại bỏ phần tử: 40
Số lượng phần tử sau khi bỏ: 2
Loại bỏ phần tử: 14
Số lượng phần tử sau khi bỏ: 1
Loại bỏ phần tử: 15
Số lượng phần tử sau khi bỏ: 0
Nothing in array a
Thêm phần tử: 57

```

```

Thêm phần tử: 57
Số lượng phần tử sau khi thêm: 1
Thêm phần tử: 2
Số lượng phần tử sau khi thêm: 2
Thêm phần tử: 79
Số lượng phần tử sau khi thêm: 3
Thêm phần tử: 22
Số lượng phần tử sau khi thêm: 4
Thêm phần tử: 90
Số lượng phần tử sau khi thêm: 5
Thêm phần tử: 73
Số lượng phần tử sau khi thêm: 6
Thêm phần tử: 9
Số lượng phần tử sau khi thêm: 7
Loại bỏ phần tử: 9
Số lượng phần tử sau khi bỏ: 6
Loại bỏ phần tử: 73
Số lượng phần tử sau khi bỏ: 5
Loại bỏ phần tử: 90
Số lượng phần tử sau khi bỏ: 4
Loại bỏ phần tử: 22
Số lượng phần tử sau khi bỏ: 3
Loại bỏ phần tử: 79
Số lượng phần tử sau khi bỏ: 2
Loại bỏ phần tử: 2
Số lượng phần tử sau khi bỏ: 1
Loại bỏ phần tử: 57
Số lượng phần tử sau khi bỏ: 0
Nothing in array a

```



- Quan sát kết quả ta thấy không có 1 thời điểm nào số phần tử trong mảng a vượt quá 7 (khi nhập n = 7).
- Khi số phần tử trong mảng còn 0 thì chương trình không thể thực hiện xóa đi 1 phần tử nữa mà phải chờ cho tới khi mảng có thêm phần tử thì mới được xóa.

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

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.

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

int x = 0;

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

```

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

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;
}

```

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

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

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

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

- Dựa vào kết quả chạy được cung cấp, có thể nhận thấy rằng giá trị của biến x không nhất quán trong suốt quá trình thực thi chương trình. *Điều này là do hai luồng, process A và process B đang truy cập và sửa đổi biến x đồng thời mà không có bất kỳ sự đồng bộ hóa nào.*
- Do đó, giá trị của x có thể không nhất quán giữa hai luồng và giá trị cuối cùng của x cũng không thể đoán trước được. Hành vi này là do điều kiện chạy đua xảy ra khi hai luồng cố gắng sửa đổi một tài nguyên được chia sẻ mà không có sự đồng bộ hóa thích hợp.
- Kết quả chạy cho thấy rằng chương trình có thể dẫn đến kết quả không mong muốn nếu không có sự đồng bộ hóa. Để tránh điều này, cần phải sử dụng các cơ chế đồng bộ hóa để đảm bảo rằng các luồng chỉ truy cập và sửa đổi các biến chia sẻ một cách an toàn.
- Trong trường hợp này, có thể sử dụng **mutex** để đồng bộ hóa quyền truy cập vào biến x. Cụ thể, mỗi khi một luồng cần truy cập hoặc sửa đổi x, nó sẽ khóa mutex trước. Sau khi hoàn tất, luồng sẽ mở khóa mutex để các luồng khác có thể truy cập x.
- Việc thêm mutex vào chương trình sẽ đảm bảo rằng mỗi luồng chỉ có thể truy cập x khi nó được khóa. Ngăn chặn các điều kiện chạy đua và giá trị của x sẽ luôn là nhất quán.

#### 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;
pthread_mutex_t mutex;

void* processA(void* mess) {
    while (1) {
        pthread_mutex_lock(&mutex);
```

```

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


int main() {
    pthread_mutex_init(&mutex, NULL);
    pthread_t pA, pB;

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

    while(1);
    return 0;
}

```

Trong đoạn code trên, chúng ta đã sử dụng mutex để đảm bảo rằng chỉ có một quy trình có thể truy cập và thay đổi biến x tại một thời điểm. Điều này đã giúp tránh các vấn đề liên quan đến việc hai quy trình cùng lúc truy cập và thay đổi biến x, dẫn đến kết quả không như mong đợi. mutex giúp bảo đảm tính nhất quán của dữ liệu khi có nhiều quy trình hoặc nhiều luồng cùng lúc truy cập vào cùng một tài nguyên.



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

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

Kết quả thực thi bài 4

## B. Bài tập ôn tập:

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.

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

int x1, x2, x3, x4, x5, x6;
int w, v, y, z, ans;

sem_t s15, s16, s23, s24, s35, s46, s57, s67;

void *process1(void *arg) {
    w = x1 * x2;
    sem_post(&s15);
    sem_post(&s16);
    pthread_exit(NULL);
}

void *process2(void *arg) {
    v = x3 * x4;
    sem_post(&s23);
    sem_post(&s24);
    pthread_exit(NULL);
}

void *process3(void *arg) {
    sem_wait(&s23);
    y = v * x5;
    sem_post(&s35);
    pthread_exit(NULL);
}

void *process4(void *arg) {
    sem_wait(&s24);
    z = v * x6;
    sem_post(&s46);
    pthread_exit(NULL);
}
```

```

void *process5(void *arg) {
    sem_wait(&s15);
    sem_wait(&s35);
    y = w * y;
    sem_post(&s57);
    pthread_exit(NULL);
}

void *process6(void *arg) {
    sem_wait(&s16);
    sem_wait(&s46);
    z = w * z;
    sem_post(&s67);
    pthread_exit(NULL);
}

void *process7(void *arg) {
    sem_wait(&s57);
    sem_wait(&s67);
    ans = y + z;
    pthread_exit(NULL);
}

int main() {
    printf("Enter the value for x1: ");
    scanf("%d", &x1);

    printf("Enter the value for x2: ");
    scanf("%d", &x2);

    printf("Enter the value for x3: ");
    scanf("%d", &x3);

    printf("Enter the value for x4: ");
    scanf("%d", &x4);

    printf("Enter the value for x5: ");
    scanf("%d", &x5);

    printf("Enter the value for x6: ");
    scanf("%d", &x6);

    pthread_t thread1, thread2, thread3, thread4, thread5, thread6,
    thread7;

    sem_init(&s15, 0, 0);
    sem_init(&s16, 0, 0);

```



```

sem_init(&s23, 0, 0);
sem_init(&s24, 0, 0);
sem_init(&s35, 0, 0);
sem_init(&s46, 0, 0);
sem_init(&s57, 0, 0);
sem_init(&s67, 0, 0);

pthread_create(&thread1, NULL, &process1, NULL);
pthread_create(&thread2, NULL, &process2, NULL);
pthread_create(&thread3, NULL, &process3, NULL);
pthread_create(&thread4, NULL, &process4, NULL);
pthread_create(&thread5, NULL, &process5, NULL);
pthread_create(&thread6, NULL, &process6, NULL);
pthread_create(&thread7, NULL, &process7, NULL);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
pthread_join(thread4, NULL);
pthread_join(thread5, NULL);
pthread_join(thread6, NULL);
pthread_join(thread7, NULL);

// Use ans
printf("Ans: %d\n", ans);

return 0;
}

```

- Thử tính tay lại với các giá trị bên dưới, ta thấy kết quả mà đoạn code đưa ra khớp với yêu cầu bài toán.

```

● ngocanh-22520077@TNANH-D53R8MB9:~$ gcc lab5_ans.c -o ans
● ngocanh-22520077@TNANH-D53R8MB9:~$ ./ans
Enter the value for x1: 2
Enter the value for x2: 5
Enter the value for x3: 3
Enter the value for x4: 7
Enter the value for x5: 1
Enter the value for x6: 9
Ans: 2100

```